TURING 图灵程序设计丛书

C程序员必读经典

原版畅销11年

「毒舌程序员」
为你揭开指针的
真实面纱

征服C指针

[日] 前桥和弥 著

吴雅明 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

前桥和弥

1969年出生，著有《彻底掌握C语言》、《Java之谜和陷阱》、《自己设计编程语言》等。其一针见血的“毒舌”文风和对编程语言深刻的见地受到广大读者的欢迎。

作者主页：
http://kmaebashi.com/

吴雅明

13年编程经验。其中7年专注于研发基于Java EE和.NET的开发框架以及基于UML 2.0模型的代码生成工具。目前主要关注的方向有：Hadoop/NoSQL、HTML5、智能手机应用开发等。

TURING 图灵程序设计丛书

征服C指针

[日] 前桥和弥 著
吴雅明 译

人民邮电出版社
北京

图书在版编目（CIP）数据

征服C指针 /（日）前桥和弥著 ; 吴雅明译. -- 北京 : 人民邮电出版社, 2013.2（2016.4重印）
（图灵程序设计丛书）
ISBN 978-7-115-30121-5

Ⅰ. ①征… Ⅱ. ①前… ②吴… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第280432号

内 容 提 要

本书被称为日本最有营养的 C 参考书。作者是日本著名的“毒舌程序员”，其言辞犀利，观点鲜明，往往能让读者迅速领悟要领。

书中结合了作者多年的编程经验和感悟，从 C 语言指针的概念讲起，通过实验一步一步地为我们解释了指针和数组、内存、数据结构的关系，展现了指针的常见用法，揭示了各种使用技巧。另外，还通过独特的方式教会我们怎样解读 C 语言那些让人“纠结”的声明语法，如何绕过 C 指针的陷阱。

本书适合 C 语言中级学习者阅读，也可作为计算机专业学生学习 C 语言的参考。

◆ 著　　　　[日] 前桥和弥
译　　　　吴雅明
责任编辑　傅志红
执行编辑　乐　馨
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷
◆ 开本：800×1000　1/16
印张：16.5
字数：333千字　　　　2013年 2 月第 1 版
印数：9 501－9 800册　　　　2016年 4 月北京第 6 次印刷
著作权合同登记号　图字：01-2012-4256号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版 权 声 明

（图灵公司感谢臧秀涛、刘建平对本书的审读）

译　者　序

在平时的工作中，我时常遇到两种人：一种是刚毕业的新人，问他们："以前学过 C 语言吗？"他们大多目光游离，极端不自信地回答说："学过，但是……"；另一种是做过几年 C 语言开发并自我感觉良好的人，他们大多可以使用指针熟练地操作字符数组，但面对菜鸟们提出的诸如"为什么数组的下标是从 0 而不是从 1 开始"这类"脑残"问题时，总是不耐烦地回答道："本来就是这样嘛。这是常识，你记住了就行！"（可本来为什么是这样的呢？）

本书的作者不是大学老师，更不是那些没有写过几行程序的学究，而是一位至今还工作在开发一线的程序员（在国内，工作了 5 年的你如果还在做"码农"，肯定会坐立不安了吧）。他带给大家的不是教科书中死板的说教，而是十多年经验沉淀下来的对无数个"脑残"问题的解答。在这本书初版面世的 11 年后，我在东京一个大型书店的 C 语言类别的书架上，依然还能看见这本书被放在一个非常醒目并且触手可及的位置上。

能从书架上挑出本书的人，我想大多都是对 C 语言指针带有"恐惧感"的程序员吧！其实所谓的"恐惧感"来源于"困惑"，而"困惑"又来自于"对知识点不够透彻的理解"。作者运用幽默风趣并且不失犀利的笔法，从"究竟什么是 C 语言指针"开始，通过实验一步一步地为我们解释了指针和数组、内存、数据结构的关系，以及指针的常用手法。另外，还通过独特的方式教会我们怎样解读 C 语言那些让人"纠结"的声明语法。

带着学习的态度，我对原著的每一个章节阅读三次以上后才开始动笔翻译。每次阅读我都会有新的收获，建议购买本书的读者不要读了一遍就将其束之高阁（甚至一遍读不下来就扔到一边）。隔一段时间再来读一遍，收获会更多。

在翻译的过程中，我身边的许多人给了我莫大的支持和鼓励。我的同事的宋岩、王红升在 C 语言方面都具有 10 年以上的编程经验，他们经常牺牲个人的休息时间帮我试读译稿，提出了诸多宝贵的意见和建议。开始翻译这本书时，我儿昀好刚出生三个月。新的生命改变了一家的生活状态，带给我们更多的是感动和欢乐。妻子葛亚文在我翻译本书的期间默默承受了产后在身体上和精神上的巨大压力，这不是一句感谢能够回报的。借此祝愿一家——四季有昀，岁月静好！

吴雅明

2012/11/26　于北京

前　　言

这是一本关于 C 语言的数组和指针的书。

一定有很多人感到纳闷："都哪朝哪代了，还出版 C 语言的书。"

C 语言确实是非常陈旧的语言，不过也不可能马上放弃对它的使用。至少在书店里，C 语言方面的书籍还是汗牛充栋的，其中专门讲解指针的书也有很多。既然如此，还有必要旧瓶装新酒吗？这才是最应该质疑的吧。

但是，每当我看到那些充斥在书店里的 C 语言入门书籍，总会怀疑这些书的作者以前根本没有使用 C 开发过大规模的系统。当然，并不是所有书的作者都这样。

指针被认为是 C 语言中最大的难点，对它的讲解，很多书都搞得像教科书一样，叙述风格雷同，让人感觉有点装腔作势。就连那些指针的练习题，其中的说明也让人厌倦。

能够炮制出这样的书籍，我想一般都得归功于那些连自己对 C 语言语法都是一知半解的作者。特别是面对那些在封面上堂堂正正地印上"第 2 类信息处理考试"①字样的书，这种感觉更加强烈。

当我还是个菜鸟的时候，也曾对数组和指针的相关语法感到非常"纠结"。

正是抱着"要是那个时候上天能让我遇见这样一本书，那可真帮了大忙"的想法，我写了这本书。

本书的内容，是基于我很久以前（1998 年 7 月）就开始在网上公开的内容：

"深入学习数组和指针"

http://kmaebashi.com/programmer/pointer.html

"当我傻呀？既然可以在网上阅读，我干嘛还买你的书？"我想对有此想法的人说："我敢打包票，绝不会让你吃亏的，请放心地拿着这本书去收款台结账吧！"因为此书在出版过程中追加

---

① 日本国内关于计算机信息处理方面的考试，主要面向计算机系统开发、维护、运用领域的初级技术人员。
——译者注

了大量的文字和插图，实际上已经比网上公开的内容丰富了许多。

另外，在阅读本书的过程中，请留心以下几点。

- 本书的读者群虽然定位于“学习过 C 语言，但是在指针的运用上遇到困难”的读者，但还是能随处可见一些高难度的内容。那是因为我也不能免俗，偶尔也喜欢把自己掌握的知识拿出来显摆一下。

  对于初学者，你完全没有必要从头开始阅读。遇到还不太明白的地方，也不要过分纠结。阅读中可以跳跃章节。对于第 0 章和第 1 章，最好还是按顺序阅读。如果认为第 2 章有点难度，你可以先去啃第 3 章。如果第 3 章也不懂，不妨尝试先去阅读第 4 章。这种阅读方式是本书最大的卖点。

- 在本书中，我会经常指出一些“C 的问题点”和“C 的不足”。可能会有一些读者认为我比较讨厌 C 语言。恰恰相反，我认为 C 是一门伟大的开发语言。倒不是因为有“情人眼里出西施”、“能干的坏小子也可爱”这样的理由，毕竟在开发现场那些常年被使用的语言中，C 语言还是有相当实力的。就算是长得不太帅，但论才干，那也是“开发现场的老油条”了。

  所以，因阅读本书而开始抱怨“C 语言真是很差劲”的读者，你即使计划了什么“去揍 Dennis Ritchie[①]之旅”，我也不会去参加的。如果有“去揍 James Gosling[②]之旅”，那还是有点心动的。哈，还是算了吧，得过且过就行啦。

在本书的写作过程中，我得到了很多人的帮助。

繁忙之中阅读大量原稿并指出很多错误的泽田大浦先生、山口修先生、桃井康成先生，指出本书网上公开内容的错误的人们，还有那些受到发布在公司内部的内容的影响而沦为“实验小白鼠”的人们，以及通过 fj.com.lang.c 和各种邮件列表进行讨论并且提供各种信息的人们，正是因为你们，本书的内容才能更加可靠。当然，遗留的错误由我来承担所有责任。

发现我的网页，并给予出版机会的技术评论社的熊谷裕美子小姐，还有给予初次写书的我很多指导的编辑高桥阳先生，如果没有他们的大力协助，这本书是不可能诞生的。

在这里，我谨向他们致以深深的谢意。

2000 年 11 月 28 日 03：33 J.S.T.

前桥和弥

① C 语言之父。本书中对他做了介绍。
② Java 语言之父。

目　录

第 0 章

本书的目标与结构——引言

0.1 本书的目标

在 C 语言的学习中，指针的运用被认为是最大的难关。

关于指针的学习，我们经常听到下面这样的建议：

> “如果理解了计算机的内存和地址等概念，指针什么的就简单了。”
> “因为 C 是低级语言，所以先学习汇编语言比较好。”

果真如此吗?

正如那些 C 语言入门书籍中提到的那样，变量被保存在内存的“某个地方”。为了标记变量在内存中的具体场所，C 语言在内存中给这些场所分配了编号（地址）。因此，大多数运行环境中，所谓的“指针变量”就是指保存变量地址的变量。

到此为止的说明，所有人都应该觉得很简单吧。

理解“指针就是地址”，可能是指针学习的必要条件，但不是充分条件。现在，我们只不过刚刚迈出了“万里长征的第一步”。

如果观察一下菜鸟们实际使用 C 指针的过程，就会发现他们往往会有如下困惑。

- 声明指针变量 `int *a;`……到这里还挺像样的，可是当将这个变量作为指针使用时，依然悲剧地写成了`*a`。

- 给出 int &a;这样的声明（这里不是指 C++编程）。
- 啥是“指向 int 的指针”？不是说指针就是地址吗？怎么还有“指向 int 的指针”，“指向 char 的指针”，难到它们还有什么不同吗？
- 当学习到“给指针加 1，指针会前进 2 个字节或者 4 个字节”时，你可能会有这种疑问：“不是说指针是地址吗？这种情况下，难道指针不应该是前进 1 个字节吗？”
- scanf()中，在使用%d 的情况下，变量之前需要加上&才能进行传递。为什么在使用%s 的时候，就可以不加&？
- 学习到将数组名赋给指针的时候，将指针和数组完全混为一谈，犯下“将没有分配内存区域的指针当做数组进行访问”或者“将指针赋给数组”这样的错误。

出现以上混乱的情形，并不是因为没有理解“指针就是地址”这样的概念。其实，真正导演这些悲剧的幕后黑手是：

- C 语言奇怪的语法
- 数组和指针之间微妙的兼容性

某些有一定经验的 C 程序员会觉得 C 的声明还是比较奇怪的。当然也有一些人可能并没有这种体会，但或多或少都有过下面的疑问。

- C 的声明中，[]比*的优先级高。因此，char *s[10]这样的声明意为“指向 char 的指针的数组”——搞反了吧？
- 搞不明白 double (*p)[3];和 void (*func)(int a);这样的声明到底应该怎样阅读。
- int *a 中，声明 a 为“指向 int 的指针”。可是表达式中的指针变量前*却代表其他意思。明明是同样的符号，意义为什么不同？
- int *a 和 int a[]在什么情况下可以互换？
- 空的[]可以在什么地方使用，它又代表什么意思呢？

本书的编写就是为了回答以上这样的问题。

很坦白地说，我也是在使用了 C 语言好几年之后，才对 C 的声明语法大彻大悟的。

世间的人们大多不愿意承认自己比别人愚笨，所以总是习惯性地认为“实际上只有极少的人才能够精通 C 语言指针”，以此安慰一下自己那颗脆弱的心。

例如，你知道下面的事实吗？

- 在引用数组中的元素时，其实 a[i]中的[]和数组毫无关系。
- C 里面不存在多维数组。

如果你在书店里拿起这本书，翻看几页后心想："什么呀？简直是奇谈怪论！"然后照原样把书轻轻地放回书架。那么你恰恰需要阅读这本书。

有人说："因为 C 语言是模仿汇编语言的，要想理解指针，就必须理解内存和地址等概念。"你可能会认为：

> "指针"是 C 语言所特有的、底层而邪恶的功能。

其实并不是这样的。确实，"C 指针"有着底层而邪恶的一面，但是，它又是构造链表和树等"数据结构"不可缺少的概念。如果没有指针，我们是做不出像样的应用程序的。所以，凡是真正成熟的开发语言，必定会存在指针，如 Pascal、Delphi、Lisp 和 Smalltalk 等，就连 Visual Basic 也存在指针。早期的 Perl 因为没有指针而饱受批评，从版本 5 开始也引入了指针的概念。当然，Java 也是有指针的。很遗憾，世上好像对此还存有根深蒂固的误解。

在本书中，我们将体验如何将指针真正地用于构造数据结构。

"指针"是成熟的编程语言必须具有的概念。

尽管如此，为什么 C 的指针却让人感觉格外地纠结呢？理由就是，C 语言混乱的语法，以及指针和数组之间奇怪的兼容性。

本书旨在阐明 C 语言混乱的语法，不但讲解了"C 特有的指针用法"，还针对和其他语言共有的"普遍的指针用法"进行了论述。

下面，让我们来看一下本书的具体结构。

0.2 目标读者和内容结构

本书的目标读者为：

- 粗略地读过 C 语言的入门书籍，但还是对指针不太理解的人
- 平时能自如地使用 C 语言，但实际对指针理解还不够深入的人

本书由以下内容构成。

- 第 1 章：从基础开始——预备知识和复习
- 第 2 章：做个实验见分晓——C 是怎样使用内存的

- 第 3 章：揭秘 C 的语法——它到底是怎么回事
- 第 4 章：数组和指针的常用用法
- 第 5 章：数据结构——真正的指针的使用方法
- 第 6 章：其他——拾遗

第 1 章和第 2 章主要面向初学者。从第 3 章开始的内容，是为那些已经具备一定经验的程序员或者已经读完第 1 章的初学者准备的。

面向初学者，第 1 章和第 2 章从“指针就是地址”这个观点开始讲解。

通过 `printf()`来“亲眼目睹”地址的实际值，应该说，这不失为理解指针的一个非常简单有效的方式。

对于那些“尝试学习了 C 语言，但是对指针还不太理解”的人来说，通过自己的机器实际地输出指针的值，可以相对简单地领会指针的概念。

首先，在第 1 章里，针对 C 语言的发展过程（也就是说，C 是怎样“沦为”让人如此畏惧的编程语言的）、指针以及数组进行说明。

对于指针和数组的相互关系，市面上多数的 C 语言入门书籍只是含混其辞地做了敷衍解释（包括 *K&R**，我认为该书是诸恶之源）。这还不算，他们还将本来已经用数组写好的程序，**特地用指针运算的方式重新编写**，还说什么“**这样才像 C 语言的风格**”。

* *K&R* 被称为 C 语言的宝典（中文版叫《C 程序设计语言（第 2 版·新版）》），在后面我们会提及这本书的背景。此书的作者之一就是 C 语言之父 Dennis Ritchie 本人。

> 像 C 语言的风格？也许是可以这么说，但是以此为由炮制出来的难懂的写法，到底好在哪里？ 哦？执行效率高？为什么？这是真的吗？

产生这些疑问是正常的，并且，**这么想是正确的**。

了解 C 语言的发展过程，就能理解 C 为什么会有“指针运算”等这样**奇怪的**功能。

第 1 章中接下来的内容也许会让初学者纠结，因为我们将开始接触到数组和指针的那些**容易让人混淆的**语法。

第 2 章讲解了 C 语言实际上是怎样使用内存的。

在这里同样采用直观的方式将地址输出。请有 C 运行环境的读者一定亲手输入例程的代码，并且尝试运行。

对于普通的局部变量、函数的参数、`static` 变量、全局变量及字符串常

量（使用" "包围的字符串）等，知晓它们在内存中实际的保存方式，就可以洞察到 C 语言的各种行为。

遗憾的是，几乎所有使用 C 语言开发的程序，运行时检查都不是非常严密。一旦出现诸如数组越界操作，就会马上引起“内存区域破坏”之类的错误。虽然很难将这些 bug 完全消灭，但明白了 C 如何使用内存之后，至少可以在某种程度上预防这些 bug 的出现。

第 3 章讲解了与数组和指针相关的 C 语言语法。

虽然我多次提到“究竟指针为什么这么难”，但是对于“指针就是地址”这个观点，在理解上倒是非常简单。出现这种现象，其实缘于 C 语言的数组和指针的语法比较混乱。

乍看上去，C 语言的语法比较严谨，实际上也存在很多例外。

对于那些平时和我们朝夕相处的语法，究竟应套用哪条规则？还有，哪些语法需要特殊对待？关于这些，第 3 章里会做彻底的讲解。

那些自认为是老鸟的读者，可以单独拿出第 3 章来读一读，看看自己以前是如何上当的。

第 4 章是实践篇，举例说明数组和指针的常用用法。如果读者理解了这部分内容，对付大部分程序应该不在话下。

老实说，对于已经将 C 语言使用得像模像样的读者来说，第 4 章中举出的例子并没有什么新意。但是，其实有些人对这些语法只是一知半解，很多时候只不过是依照以前的代码“照猫画虎”罢了。

阅读完第 3 章后去读第 4 章，对于那些已经能够熟练使用的写法，你也会惊呼一声：“原来是这个意思啊！”

第 5 章中，解说指针真正的用法——数据结构的基本知识。

前四章中的例子，都是围绕 C 语言展开的。第 5 章里则会涉及其他语言的指针。

无论使用哪种语言编程，“数据结构”都是最重要的。使用 C 语言来构造数据结构的时候，结构体和指针功不可没。

“不仅仅对于 C 语言的指针，连结构体也不太明白”的读者，务必不要错过第 5 章。

第 6 章中，对到此为止还没有覆盖到的知识进行拾遗，并且为大家展示一些可能会遇到的陷阱以及惯用语法。

和类似的书籍相比，本书更加注重语法的细节。

提起语法，就有“日本的英语教育不偏重语法”，以至于给人“就算不明白也没有什么”的印象。确实是这样，我们早在不懂“サ行”怎样变形之前，就已经会说日语了。

但是，C 语言可不是像日语那样复杂的自然语言，它是一门编程语言。

单纯地通过语法来解释自然语言是非常困难的。比如，日语“いれたてのおちゃ”，利用假名汉字变换程序只能变换成“淹れたてのお茶”（中文意思是‘沏好的茶’），尽管如此，同样通过假名变换程序，“いれたてのあついおちゃ”竟然也可以变换成“入れた手の厚いお茶”（中文意思是‘沏好的热腾腾的茶’）[①]。编程语言最终还是通过人类制订的语法构成的，它要做到让编译器这样的程序能够解释。

“反正大家都是这么写，我也这么写，程序就能跑起来。”

这种想法，让人感觉有点**悲哀**。

我希望不仅是初学者，那些已经积累了一定经验的程序员也能阅读本书。通过深入理解 C 的语法，可以让我们真正领会直到今天还像“口头禅”一样使用的那些程序惯用写法。

无论如何，让我们做到“知其然知其所以然”，这样有利心理健康，不是吗？

---

① 中文里也有类似的例子，如：他是先知。→他是先知道那件事的人。——译者注

第 1 章

从基础开始——预备知识和复习

1.1 C是什么样的语言

1.1.1 比喻

在 Donald C. Gause 和 Gerald M. Weinberg 合著的《你的灯亮着吗？》[1]一书*中，有这样一节（根据需要，我做了必要的删减）。

> 某计算机制造商开发了一种新型打印机。
>
> 技术小组在如何保证打印精度的问题上非常苦恼，每次进行新的测试时，工程师都不得不花很长的时间测量打印机的输出结果来追求精确性。
>
> 丹（Dan Daring）是这个小组中最年轻但或许是最聪明的工程师。他发明了一种工具，即每隔 8 英寸就在铝条上嵌上小针。使用这个工具，可以很快地找到打印机输出位置的误差。
>
> 这个发明显著地提高了生产效率，丹的上司非常高兴，提议给丹颁发一个公司的特别奖赏。他从车间里拿了这个工具，带回办公室，这样他写报告的时候还可以仔细地研究一下。
>
> 这个上司显然还用不惯这个工具，当他把这个工具放在桌子上的时候，将针尖朝上了。更不幸的是，当丹的上司的上司友好地坐到桌角上，打算谈谈给丹颁发奖励时，部门内的所有人都听到了他痛苦的尖叫声——他的屁股上被扎了两个相距 8 英寸的孔。

C 语言就恰如这个工具。也就是说，它是一门

* 这本书的副标题为“发现问题的真正所在”，它通过一些趣闻轶事来告诉世人“不要急于寻找问题的答案，而是应该先去考虑当前的问题是什么”。

- 为了解决眼前问题，由开发现场的人发明的，
- 虽然使用方便，
- 但看上去不怎么顺眼，
- 如果不熟悉的人糊里糊涂地使用了它，难免会带来“悲剧”的语言。

1.1.2 C 的发展历程

众所周知，C 原本是为了开发 UNIX 操作系统而设计的语言。

如此说来，好像 C 应该比 UNIX 更早问世，可惜事实并非如此，最早的 UNIX 是用汇编来写的。

因为厌倦了总是苦哈哈地使用汇编语言进行编程，UNIX 的开发者 Ken Tompson 开发了一种称为“B”的语言。B 语言是 1967 年剑桥大学的 Martin Richard 开发的 BCPL（Basic CPL）的精简版本。BCPL 的前身是 1963 年剑桥大学和伦敦大学共同研究开发的 CPL（Combined Programming Lanugage）语言。

B 语言不直接生成机器码，而是由编译器生成栈式机（Stack Machine）用的中间代码，中间代码通过解释器（interpreter）执行（类似 Java 和早期的 Pascal）。因此，B 语言的执行效率非常低，结果，在后来的 UNIX 开发过程中人们放弃了使用 B 语言。

在这之后的 1971 年，Ken Tompson 的同事 Dennis Ritchie 对 B 语言做了改良，追加了 char 数据类型，并且让 B 语言可以直接生成 PDP-11*的机器代码。曾经在很短的时间内，大家将这门语言称为 NB（New B）。

* 现在已经不存在的 DEC（美国数字设备公司）生产的微型电脑。

之后，NB 改称为 C 语言——这就是 C 语言的诞生。

后来，主要是为了满足使用 UNIX 的程序员的需要，C 语言一边接受来自各方面的建议，一边**摸着石头过河**般地进行着周而复始的功能扩展。

1978 年出版了被称为 C 语言宝典的 *The C Programming Language* 一书。

此书取了两位作者（Brian Kernighan 和 Dennis Ritchie）的姓氏首字母，简称为 *K&R*。在后面提到的 ANSI 标准制定之前，此书一直作为 C 语言语法的参考书被人们广泛使用。

听说这本书在最初发行的时候，Prentice-Hall 出版社制订了对于当时存在的 130 个 UNIX 站点平均每个能卖 9 本的销售计划（相比 *Lift With UNIX*[2]）。

当然了，哪怕是初版 *K&R* 的销售量，也以 3 位数的数量级超过了 Prentice-Hall 出版社最初的销售计划*。原本只是像“丹的工具”一样为了满足自用的 C 语言，历经坎坷，最终成为全世界广泛使用的开发语言。

* 我手头这本 *K&R* 是在 1997 年 5 月 1 日出版的第二版（翻译修订版），已经是第 211 次印刷了。这个行业的图书能有这样的业绩，确实惊人。



补充　B 是什么样的语言？

在 C 语言的入门书籍中，经常提到 C 是 B 语言的进化版本。但几乎所有的书对 B 语言的介绍都只有这么多，没有具体说明 B 语言究竟是一门什么样的语言。

正如前面描述的那样，B 是在虚拟机上运行的、解释型开发语言。B 语言没有像 Java 那样想要去实现“到处运行”的宏伟目标，它只是因为当时运行 UNIX 的 PDP-7 硬件环境的限制，而只能采用解释器这样的实现方式。

B 是“没有类型”的语言。虽然 C 里面有 `char`、`short`、`int`、`float` 和 `double` 等很多的数据类型，但是 B 使用的类型只有 `word`（你可以认为和 `int` 差不多）。作为本书的主题——指针，在 B 里面和整数一样使用。因为无论怎么说，指针无非就是内存中的地址，对于机器来说，它是可以和整型同样对待的（关于这点，在本章中会详细说明）。

NB 是具有类型概念的语言。为了把指针和整数纠葛不清的 B 移植到 NB，Dennis Ritchie 在指针运用的设计上下了很大的工夫。C 的指针让人感觉很难理解，可能也有这方面的原因。

关于 B 语言，如果你想知道更详细的内容，可以浏览 Dennis Ritchie 的网页：http://cm.bell-labs.com/cm/cs/who/dmr/index.html 中的“Users' Reference to B”[3]等内容。

1.1.3　不完备和不统一的语法

C 语言是开发现场的人们根据自身的需要开发出来的语言，所以具备极高的实用性。但反过来从人类工程学的角度来看，它就不是那么完美了。

比如：

```
if (a = 5) {  ←— 本来应该写成==的地方却写成了=
```

相信大家都犯过这样的错误吧。

在日语键盘上，“-”和“=”在同一按键上，因此经常会发生下面的问题：

```
for (i = 0; i < 100; i++)  ←— 忘了和[shift]键一起按下
```

就连这样的错误，编译器也往往无法察觉。现在的编译器倒是可以给出警告，可是早期的编译器对这样的错误是完全忽略的。

使用 switch case 的时候，也经常发生忘了写 break 的错误。

幸运的是，如今的编译器，对于容易犯的语法错误，在很多地方可以给我们警告提示。因此，不但不能无视这些编译器的警告，相反应该提高编译器的警告级别，让编译器替我们指出尽可能多的错误*。

* 尽管如此……假设无视了有返回值的函数返回的值，lint 会给出一个警告。为了消除这个讨厌的警告，你特地使用 (void)printf(...) 打印返回的值，这么干是不是就有点过了？C 原本就是“本性恶劣”的语言，警告级别过高，会否定一些既存的程序写法，反而带来不好的后果。此外，如果最大限度地提高警告级别，有些应用程序自身包含的头文件也会引起警告。总之，这些问题还是比较麻烦的。

换句话说，如果编译器向我们提示错误或者警告，不应该以怨报德：“什么呀，这个混蛋！”相反应该奉上一句感谢：“谢谢你，编译器先生！”然后去认真地清除眼前的 bug。

> 要　点
> **提高编译器的警告级别。**
> **不可无视或者制止编译器的警告。**

C 语言是在使用中成长起来的语言。因此，由于很多历史原因遗留了一些“奇怪的”问题。具有代表性的有位运算符“&”和“|”的优先顺序问题。

通常，如“==”的比较运算符的优先级要低于那些做计算的运算符。因此，

```
if (a < b + 3)
```

这样的条件表达式中，虽然可以不使用括号来写，但是当使用了位运算符的时候，就行不通了。

想要进行“将 a 和 MASK 进行按位与运算后的结果，再和 b 做比较运算”，

```
if (a & MASK == b)
```

按照上面的写法，因为&运算符的优先级低于==运算符，所以被解释成了下面这样：

```
if (a & (MASK == b))
```

这是因为在没有“&&”和“||”运算符的时代，使用“&”和“|”来代替而留下的后遗症。

1.1.4　ANSI C

即使在 *K&R* 出版之后，C 仍在不断地扩展。

比如关于结构体的一次性赋值，在*K&R*的初版里面并没有记述，其实这个功能在*K&R*出版之前，就已经在Dennis Ritchie的C编译器里实现了。从某种意义上来说，*K&R* 的第一版刚出版就已经过时了。这在计算机图书界是常有的事。

另外，*K&R* 的记述也不一定就是严密的，由于运行环境的不同，程序运行也存在差异。

鉴于这些原因，经过一番纷争，终于在1989年，ANSI（American National Standard Institute，美国国家标准学会）通过了C语言的标准规范，这就是通常被称为ANSI C的C语言。目前使用的C程序，大部分都是基于ANSI C编写的。

顾名思义，ANSI是美国的标准。难道不存在C语言的国际标准吗？那是不可能的。ANSI C后来被ISO采用，目前C的真正标准应该是ISO的C。另外，原始的ANSI标准说明书的章节编号和ISO不同。

ISO-C标准的名称为“ISO-IEC 9899-1990”，当然这是用英语命名的。JIS（日本工业标准）标准（JIS X3010）原样采用了ISO-C标准，所以英语不太好的人（比如我），也可以获取日语版的标准文档。①

从日本工业标准协会可以得到JIS X3010标准，以及JIS X3010的手册。另外，《信息处理：编程语言篇》里面也包含了这个标准*，此书可以从书店订购。如果是那些大型的书店，也许在书架上就能发现这本书。

* 2001年，JIS手册全面修订，删除了《信息处理：编程语言篇》中已经存在的内容。如果在所有的标准中都记录相同的信息会引起不必要的麻烦。

在本书后面的内容中，提到的“标准”都是指JIS X3010。

1.1.5 C的宝典——*K&R*

之前已经介绍过，Brian Kernighan和Dennis Ritchie（C语言之父）合著的*The C Programming Language*被称为*K&R*。在制订ANSI C之前，*K&R*是C语言语法的使用标准。

人们把ANSI之前的C称为“K&R C”，这可能会引起一些误解。但无论怎样，制订了ANSI C标准之后，追随ANSI C的*K&R*就紧跟着出版了第2版[4]。

在本书中，提起ANSI C之前的C，我们还是尊重事实，称之为“ANSI C之前的C”。

① 国内读者可参考中国国家标准GB/T 15272-94，它即为ISO-IEC 9899-1900的中文翻译版。——译者注

此外，本书中提及 *K&R* 时，是指日语版的《编程语言 C》的第 2 版修订版*。

* 最初的 *K&R* 第 2 版，因为翻译质量的问题，恶评如潮。重新翻译后再次出版了修订版（原书相同）。第 2 版旧翻译版的封面为绿色，新翻译版的封面为白色。[1]

顺便提一下，通过

http://www.cs.bell-labs.com/cm/cs/cbook/index.html

可以看到 *K&R* 的网页，各语种的 *K&R* 封面排列在一起，颇为壮观。



补充　新的 C

C 语言的功能扩展并没有随着 ANSI C 的发布而停下脚步。ISO 通过 ISO C9X 这个代号名称，计划制定具备更多扩展功能的 C 语言规范。

从 ISO C9X 这个代号名称可以看出来，大家都很期待在 20 世纪 90 年代完成新标准的制订，标准文档封面上的日期是 1999 年 12 月 1 日——好险呀!!

作为 ISO C99 的扩展功能，除了提供对复数类型的支持之外，还包括“可以用变量定义本地数组变量的元素个数”、“将数组作为结构体的成员进行声明时元素个数可以不定义（只需写[]）”等功能。这些功能看上去都和本书的主题有重合的部分。

尽管如此，这个“新的 C”在今后究竟能使用多久，我们现在不得而知。所以，本书不参照 ISO C99。

1.1.6　C 的理念

ANSI C 标准，附有 Rationale（理论依据）。

可以通过下面的地址在线获取 Rationale。

ftp://ftp.uu.net/doc/standards/ansi/x3.159-1989/

Rationale 中有“keep the spirit of C”（保持 C 的精神）一节，关于“C 的精神”是这样介绍的：

❶ 请信任程序员（Trust the programmer）
❷ 不要阻止程序员去做需要做的工作（Don’t prevent the programmer

[1] 中文版请参考《C 程序设计语言（第 2 版・新版）》。——译者注

from doing what needs to be done）
❸ 保持语言的小巧和简单（Keep the language small and simple）
❹ 为每一种操作只提供一种方法（Provide only way to do an operation）
❺ 就算不能保证可移植性，也要追求运行效率（Make it fast, even if it is not guaranteed to be portable）

前面两点最重要——这样不负责任的话它还真敢说！

C是危险的语言，其中随处可见那些可以让丹的上司的上司在不留心的时候，屁股上扎两个洞的陷阱。

尤其是，在几乎所有的C语言实现中，运行时的检查总是不充分的。比如，数组越界写入的时候，有些语言是可以当场给出错误提示的（如Java），但是在C的大部分处理中，总是悄悄地将数据写入，从而破坏了完全不相关的内存区域。

C是抱着“**程序员万能**”的理念设计出来的。在C的设计中，优先考虑的是：

- 如何才能简单地实现编译器（而不是让使用C的人们能够简单地编程）
- 如何才能让程序员写出能够生成高效率执行代码的程序（而不是考虑优化编译器，使编译器生成高效率的执行代码）

而安全性问题被完全忽略了。但无论怎样，C语言原本就是“仅仅为了自己使用”而开发出来的语言。

幸运的是，如今的操作系统可以替我们停止那些已经明显出现奇怪动作的应用程序。在UNIX环境下，可以提示“Segmentation fault”、“Bus Error”等信息。Windows也可以向我们提示“当前应用程序正在进行异常操作，所以需要强制关闭”这样的信息。

同样，这个时候也不能抱有“什么呀，这个混蛋！”的想法，而是应该奉上一声感谢：“谢谢，操作系统先生！”然后埋头开始调试工作。

虽说操作系统可以在应用程序发生明显错误的时刻为我们关闭应用，但是在超出数值型的字节长度或数组的边界写入数据的时候，追踪bug还是很困难的，因为这些错误症状很少能马上显现。

本书第2章将说明C是怎样使用内存的。理解了这一点，对消灭这些不易发现的bug大有裨益。

要 点

很幸运，操作系统可以帮助我们停止应用程序。

糟糕的是，如果操作系统不能替我们终止执行应用程序，就会上演内存区域被破坏的悲剧。

1.1.7 C 的主体

我在这里想考考你。

下面的单词中，哪些是 C 语言中规定的保留字。

```
if  printf  main  malloc  sizeof
```

答案是 `if` 和 `sizeof`。

“`printf` 和 `malloc` 不必多说，连 `main` 也不是 C 的保留字吗？”有这样想法的读者，请查一查手头的 C 语言参考书。相信大部分的 C 入门书籍中都有 C 语言保留字的列表。

C以前的很多语言，把输入输出作为语言自身功能的一部分。比如在Pascal中与 C 的 `printf()`的功能相当的，是使用 `write()`这样的标准规范。它在Pascal 的语法规则中受到了特别对待*。

* 根据 JIS X3008 6.6.4.1 的备注内容，“标准规范或标准函数不一定遵从标准和函数的一般规则”。

相对这种方式，C 语言将 `printf()`这样的输入输出功能从语言的主体部分分离出来，让它单纯地成为库函数。对于编译器来说，`printf()`函数和其他由普通程序员写的函数并没有什么不同*。

* 偶尔也有可以帮我们检查 `printf()`的参数个数的编译器……

从程序员的角度来看，`printf()`操作一下子就完成了。其实为了完成这个操作，需要在幕后做诸如向操作系统进行各种各样的请求等非常复杂的处理。C 语言并没有把这种复杂的处理放在语言主体部分，而将它们全部规划在函数库中。

很多编译型的语言会将被称为“run-time routine”（运行时例程）的机器码“悄悄地”嵌入到编译（链接）后的程序中，输入输出这样的功能就是包含在 run-time routine 之中的。C 语言基本上没有必须要“悄悄地”嵌入运行时的复杂功能*。由于稍微复杂一点的功能被全部规划到了库中，程序员只需要去显式地调用函数。

* 当初基于 PDP-11 的应用程序，处理 32 位的乘除运算，以及函数入口和出口的运行时是被“悄悄地”嵌入的。

诚然，这种方式有它的缺点，但是也有它的优点。正因为这个优点，才可能使 C 语言的程序开发和学习变得容易一些。

1.1.8　C 是只能使用标量的语言

对于**标量**（scalar）这个词，大家可能有些陌生。

简单地说，标量就是指 char、int、double 和枚举型等数值类型，以及指针。相对地，像数组、结构体和共用体这样的将多个标量进行组合的类型，我们称之为**聚合类型**（aggregate）。

早期的 C 语言**一度只能使用标量**。

经常听到初学者有以下的提问：

```
if (str == "abc")
```

> 这样的代码为什么不能执行预期的动作呢？确实已经将“abc”放到了 str 中，条件表达式的值却不为真。这是为什么？

对于这样的疑问，通常给出的答案是“这个表达式不是在比较字符串的内容，它只是在比较指针”，其实还可以给出另外一个答案：

> 字符串其实就是 char 类型的**数组**，也就是说它不是标量，当然在 C 里面不能用==进行比较了。

C 就是这样的语言，一门“不用说对于输入输出，就连数组和结构体也放弃了通过语言自身进行整合利用”的语言。

但是，如今的 C（ANSI C）通过以下几个追加的功能，已经能够让我们整合地使用聚合类型了。

- 结构体的一次性赋值
- 将结构体作为函数参数值传递
- 将结构体作为函数返回值返回
- auto 变量的初始化

当然，这些都是非常方便的功能，如今已经可以积极地使用了（不如说应该去使用）。可是在早期的 C 语言里，它们是不存在的。为了理解 C 语言的基本原则，了解早期的 C 语言也不是什么坏事。

特别要提出来的是，即使是 ANSI C，也还不能做到对数组的整合利用。将数组赋值给另外一个数组，或者将数组作为参数传递给其他函数等手段，在 C 语言中是不存在的。

但是，因为结构体是可以被整合利用的，所以在实际的编程中，应该积极地使用其可用的功能。直到现在，还经常能看到使用 `memcpy()`来进行结构体一次性赋值的例子，真是做无用功。如果想要复制结构体，还是让我们使用结构体一次性赋值这个功能吧。

1.2　关于指针

1.2.1　恶名昭著的指针究竟是什么

关于“指针”一词，在 *K&R* 中有下面这样的说明（第 5 章“指针和数组”的开头部分）：

> 指针是一种保存变量地址的变量，在 C 中频繁地使用。

其实在表达上，这样的说明是有很大问题的。总会让人感觉，一旦提起指针，就要把它当作变量的意思。实际上并非总是如此。

此外，在 C 语言标准中最初出现“指针”一词的部分，有这样一段话：

> **指针类型**（pointer type）可由函数类型、对象类型或不完全的类型派生，派生指针类型的类型称为引用类型。指针类型描述一个对象，该类对象的值提供对该引用类型实体的引用。由引用类型 T 派生的指针类型有时称为“（指向）T 的指针”。从引用类型构造指针类型的过程称为“指针类型的派生”。这些构造派生类型的方法可以递归地应用。

这段话的内容也许会让你一头雾水*。那就让我们先关注第一句话吧，那里出现了“指针类型”一词。

* 既然是标准，那总要有点标准的范儿吧。

提到“类型”，立刻会让人想起“`int` 类型”、“`double` 类型”等。同样，在 C 语言中也存在“指针类型”这样的类型。

“指针类型”其实不是单独存在的，它是由其他类型**派生**而成的。以上对标准内容的引用中也提到“由引用类型 T 派生的指针类型有时称为‘（指向）T 的指针’”。

也就是说，实际上存在的类型是“指向 `int` 的指针类型”、“指向 `double` 的指针类型”。

因为“指针类型”是类型，所以它和 int 类型、double 类型一样，也存在“指针类型变量”和“指针类型的值”。糟糕的是，“指针类型”、“指针类型变量”和“指针类型的值”经常被简单地统称为“指针”，所以非常容易造成歧义，这一点需要提高警惕*。

* 至少本书还是尽力将这些说法进行区别的，但有时候，无论怎么写也做不到自然地表述想要表达的意思，最后只好投降……非常抱歉。

要 点

先有“指针类型”。

因为有了“指针类型”，所以有了“指针类型的变量”和“指针类型的值”。

比如，在 C 中，使用 int 类型表示整数。因为 int 是“类型”，所以存在用于保存 int 型的变量，当然也存在 int 型的值。

指针类型同样如此，既存在指针类型的变量，也存在指针类型的值。

因此，几乎所有的处理程序中，所谓的“指针类型的值”，实际是指**内存的地址**。

变量的内容是保存在内存的某个地方的，“某个地方”的说法总是会让人产生困惑，因此，就像使用“门牌号”确定“住址”一样，在内存中，我们也给变量分配“门牌号”。在 C 的内存世界里，“门牌号”被称为“地址”。

为了帮助理解这一点，还是写一个程序来验证一下。

1.2.2 和指针的第一次亲密接触

下面我们通过实际编程来尝试输出指针的值（参照代码清单 1-1）。

代码清单 1-1 pointer.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int     hoge = 5;
 6:      int     piyo = 10;
 7:      int     *hoge_p;
 8:
 9:      /*输出每个变量的地址*/
10:      printf("&hoge..%p\n", &hoge);
11:      printf("&piyo..%p\n", &piyo);
12:      printf("&hoge_p..%p\n", &hoge_p);
13:
14:      /*将 hoge 的地址赋予 hoge_p*/
15:      hoge_p = &hoge;
```

```
16:     printf("hoge_p..%p\n", hoge_p);
17:
18:     /*通过 hoge_p 输出 hoge 的内容*/
19:     printf("*hoge_p..%d\n", *hoge_p);
20:
21:     /*通过 hoge_p 修改 hoge 的内容*/
22:     *hoge_p = 10;
23:     printf("hoge..%d\n", hoge);
24:
25:     return 0;
26: }
```

下面是我的环境(FreeBSD 3.2-RELEASE 和 gcc version 2.7.2.1)里输出的结果。

```
&hoge..0xbfbfd9e4
&piyo..0xbfbfd9e0
&hoge_p..0xbfbfd9dc
hoge_p..0xbfbfd9e4
*hoge_p..5
hoge..10
```

第 5~7 行声明了 int 类型变量 hoge、piyo 和“指向 int 的指针”类型的变量 hoge_p。如果理解 hoge_p 的声明有困难，不妨先单纯地将它理解成“指向 int 的指针”类型的变量（请参照本节的补充内容）。

int 类型的变量 hoge 和 piyo，在声明的同时分别被初始化为 5 和 10。

在第 10~12 行，使用**地址运算符&**，输出各变量的地址。在我的环境中，变量在内存中保存成下面这样（请参照图 1-1）。

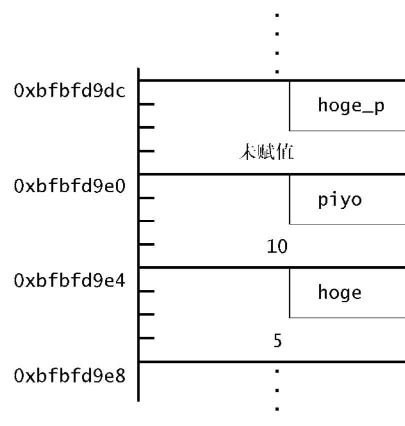


图 1-1　变量的保存状况

总觉得在我的环境中，变量是按照声明的逆向顺序保存在内存中的。可能会让人感觉有些奇妙，其实这是**常见的现象**，不要太在意。

> 要　点
> **变量不一定按照声明的顺序保存在内存中。**

前面曾经提到，因为存在“指针类型”，所以存在“指针类型的变量”和“指针类型的值”。这里输出的“地址”，是指“指针类型的值”。

另外，以上的例子在使用 printf()输出指针的值时，使用了参数%p。很多人都使用过%x 这样的参数。遗憾的是，这种使用方式是**错误**的。关于这点的解释，请参照 1.2.3 节。

在第 15 行，将 hoge 的地址赋给指针变量 hoge_p。因为 hoge 的地址是 0xbfbfd94e4，这时内存变成图 1-2 所示的状态。

像这样，指针变量 hoge_p 保存了另外一个变量 hoge 的地址，我们认为“hoge_p 指向 hoge”。

此外，对 hoge 变量实施&运算得到“hoge 的地址”。有时候也称“hoge 的地址”的值为“指向 hoge 的指针”（此时的“指针”指的是“指针类型的值”）。

在我的环境里，变量是按照声明的逆向顺序保存在内存中的。根据不同的环境，内存中变量位置的顺序可能有所不同，纠结于究竟 hoge、piyo 和 hoge_p 以什么样的顺序排列是没有意义的。图 1-2 也可以用图 1-3 的表现方式。

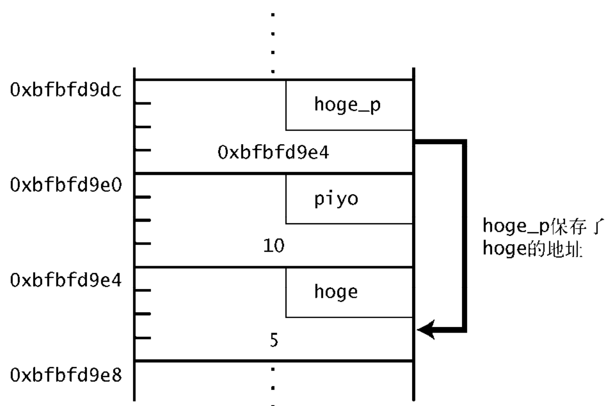


图 1-2　将指向 hoge 的指针的值赋给 hoge_p

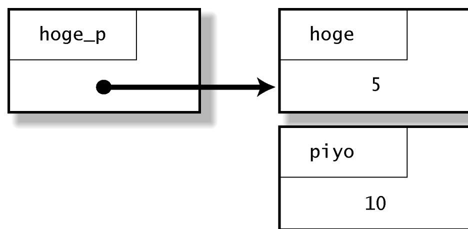


图 1-3　图 1-2 的另一种表现方式

上图更能直接地表现“hoge_p 指向 hoge”这个含义。

在第 19 行，使用**解引用***，“顺藤摸瓜”输出 hoge 的值。

在指针前面加上*，可以表示指针指向的变量。因为 hoge_p 指向 hoge，所以*hoge_p 等同于 hoge。一旦要求输出*hoge_p，就会输出 hoge 中保存的值 5。

因为*hoge_p 和 hoge 表示同一个事物，通过*hoge_p 输出 hoge 的值之外，还可以赋值。在第 22 行，通过将 10 赋给*hoge_p，修改了 hoge 的值。在第 23 行输出 hoge 的值，运行结果为 10。

指针的基本知识就介绍到这里。以下是整理出的要点。

要　点

- ❑ **对变量使用&运算符，可以取得该变量的地址。这个地址称为指向该变量的指针。**
- ❑ **指针变量 hoge_p 保存了指向其他变量的地址的情况下，可以说“hoge_p 指向 hoge”。**
- ❑ **对指针变量运用*运算符，就等同于它指向的变量。如果 hoge_p 指向 hoge，*hoge_p 就等同于 hoge。**



补充　**关于十六进制**

在说明地址概念的时候，世上的 C 语言入门书籍中经常使用“门牌号 100”这样极其小的十进制值。

确实，对于初学者来说，可能这样更容易入门。但是本书偏执地使用了十六进制来说明。这是因为，如果想要了解地址的真正面目，把地址实际地表示出来才是最好的方式。本书例程中输出的所有地址，全部是通过

我的环境实际运行程序后获得的。

对于那些对指针还是不太明白的读者来说，一定也要像我这样将例程实际地敲一遍，然后通过自己的环境确认一下究竟会输出什么东东。当然，通过你自己的环境输出的地址肯定和我的环境中输出的不同，但是其中的原理是一样。

哦？你说你不懂十六进制？——不好意思，你应该事先学习一下这方面的知识。



补充 混乱的声明——如何自然地理解声明?

通常，C 的声明像

```
int hoge;
```

这样，使用“**类型 变量名;**”的形式进行书写。

可是，像“指向 int 的指针”类型的变量，却要像下面这样进行声明:

```
int *hoge_p;
```

似乎这里声明了一个名为*hoge_p 的变量，而实际上，这里声明的变量是 hoge_p，hoge_p 的类型是“指向 int 的指针”。

因为这种声明方式不太好理解，所以有人提出将*靠近类型这一侧进行书写，如下：

```
int*  hoge_p;
```

的确，这种书写方式符合“**类型 变量名;**”的形式。但是在同时声明多个变量的情况下就会出现破绽：

```
/*声明两个“指向 int 的指针”？——其实不是*/
int*  hoge_p, piyo_p;
```

此外，数组也是 C 的一种类型，比如

```
int hoge[10];
```

这样的写法，就不符合“**类型 变量名;**”的形式。

说一些题外话，Java 在声明“int 的数组”时，通常写成

```
int[] hoge;
```

* Java 可以通过在使用 new 进行实例化的时候定义数组元素的个数，所以这里没有元素个数的声明。

的形式*，这样好像是符合“类型 变量名;”的形式。至少在这一点上，Java 的语法比起 C 显得更为合理。可是，Java 为了让 C 程序员更容易地将程序向 Java 移植，竟然也兼容 int hoge[]这样的写法。这种不伦不类的做法倒还真像 Java 的风格。

我们换个角度考虑问题，对于

```
int *hoge_p;
```

这个声明，因为当 hoge_p 指向 hoge 的时候，*hoge_p 和 hoge 可以同等地使用，所以有人可能会产生下面的想法。

> 你们看，一旦在 hoge_p 之前追加*，就可以和 int 变量 hoge 同样使用呢。也就是说，这个声明意味着 hoge_p 之前追加上*后成为 int 类型了。

这种思考方式，确实也有它一定的道理（比如数组也同样可以这么说），那么，如果写成

```
int *&hoge;
```

这样，hoge 是可以作为 int 类型的变量来声明的吧？尝试一下就会明白，这里会发生一个语法错误。

其次，在声明中出现 const 的时候，这种观点也会出现破绽（表达式中是不可以出现 const 的），声明指向函数的指针时同样会出现问题。

以我的经验来看，一切关于“如果这样考虑，是不是就可以很自然地解释 C 的声明了？”的尝试都是徒劳的。为什么这么说，因为 C 语言的语法本来就是不自然、奇怪而又变态的。

在第 3 章会详细地说明 C 的声明语法。姑且带着问题继续往下阅读吧。



补充

关于 int main(void)

在 C 语言标准中，关于 main()函数的使用只有如下两种方式：

```
int main(int argc, char *argv[])
```

或者

```
int main(void)
```

尽管如此，还是可以在一些入门书籍中遇到

```
void main(void)
```

这样的写法，**这是错误的**。确实，就算是这么写，很多程序也能动起来。但是在有些环境下，编译器可能会报告一些警告信息。

main 函数返回一个 int 类型的值，因此在处理的最后必须有 return（现在的很多编译器都会提示没有 return 的警告）。

本书所有例程的 main 函数的末尾都写了 return 0;。

返回 0 表示通知运行环境程序“正常结束”。



题外话　hoge 是什么？

本书的例程中，经常使用 hoge 或 piyo 作为变量的名称。

这是啥？很多人会有这样的疑问。在日本，hoge 这个名字使用非常广泛。在为变量和文件的取名感到苦恼的时候，大家经常使用 hoge 这个词。

通常都会给变量取一个有意义的名字，但因为本书是单纯讲解 C 语法的书，所以很多地方使用了没有实际意义的单词。当然了，就算使用了“a”、“b”这样的变量名称，编译器也不会抱怨什么，但是这种一个字母的变量名用在面向初学者的书中，似乎不太合适。

为了也能明确地表示那些没有意义的变量，我们使用具有 4 个字母的 hoge。

谁也不知道是哪个大侠最先使用 hoge 这个单词的。目前最有力的说法是，在 20 世纪 80 年代前半期，hoge 在日本各地被同时频繁地使用起来，详细的说明请参见：

“关于 hoge 的网页”

http://kmaebashi.com/programmer/hoge.html*

在美国，和 hoge 这个单词一样，foo 和 bar 等单词经常被使用。偶尔可以在 OS 的操作手册等资料中看见它们的身影。

* 原始网页已经取消不能访问了。承蒙作者吉田先生的允诺，此网页的内容已转载入我的网页中。

1.2.3　指针和地址之间的微妙关系

在本章 1.2.1 节中，有下面一句话：

> 几乎所有的处理程序中，所谓的“指针类型的值”，实际是指内存的**地址**。

对于这句话，有人也许会产生下面的疑问。

【常见疑问之 1】

归根结底，指针就是地址，地址就是内存中被分配的“门牌号”。所以，指针类型和 int 类型应该是一回事吧。

实际上，从某种意义来看，这种认识也不无道理。

在 C 语言前身的 B 语言中，指针和整数是没有区别的。此外，虽然我们经常使用 printf()和%p 来表示指针，实际上包括我的运行环境在内，使用%x 也可以很好地表示地址。对不太擅长十六进制的人来说，通过使用%d，也能利用十进制的方式来确认地址的内容。

很可惜，这里说的运行环境并不具有普适性。其实在很多的运行环境中，int 类型和指针类型的长度并不相同，此外，由于 Intel 8086 的功能限制，在直到最近还被广泛使用的 MS-DOS 中，是通过将 16 位的值分成两组来表示 20 位的地址的[1]。

还有——不，还是先回答下一个问题吧。

【常见疑问之 2】

指针就是地址吧。那么，指向 int 的指针也好，指向 double 的指针也好，它们有什么不一样吗？有必要去区分它们吗？

在某种意义上，这种说法也有一定道理。

对于大部分的运行环境来说，当程序运行时，不管是指向 int 的指针，还是指向 double 的指针，都保持相同的表现形式（偶尔也会有一些运行环境，它们对于指向 char 的指针和指向 int 的指针有着不一样内部表示和位数）。

不仅如此，ANSI C 还为我们准备了“可以指向任何类型的指针类型”——void*类型。

```
1:    int  hoge = 5;
2:    void  hoge_p;
3:
4:    hoge_p = &hoge;    ←这里不报错

5:    printf("%d\n",  hoge_p); / 打印输出 hoge_p 指向的变量的值 /
```

以上代码中的第 4 行是不会报错的。

① 8086 是分段寻址的，具体来说是指一个物理地址由**段地址**（segment selector）与**偏移量**（offset）两部分组成，长度各是 16 位。其中段地址左移 4 位（即乘以 16）与偏移量相加即为物理地址。——译者注

但是，像第 5 行这样在 hoge_p 前附加*……在我的环境里会出现下面的警告：

```
warning: dereferencing 'void *' pointer
invalid use of void expression
```

只需稍微考虑一下，就知道出现这样的错误是意料之中的。如果仅仅告之内存地址，却没有告之在那个地址上保存的数据类型，当然是不能取出值来的。

如果将上面的第 5 行修改成下面这样，不但可以顺利地通过编译，甚至可以正常地运行。

```
5:    printf("%d\n", *(int*)hoge_p); /*将 hoge_p 强制转换成 int* */
```

这里通过将“所指类型不明的指针”hoge_p 强制转型成“指向 int 的指针”，来告之编译器类型信息，由此可以取出 int 类型的值。

但每次都这样写是比较繁琐的，不妨事先写成以下的声明：

```
int *hoge_p;
```

因为**编译器可以记住**“hoge_p 是指向 int 的指针”，所以只需要简单地在 hoge_p 前面添加*，就可以通过指针间接取值。

之前也提到，在大部分的运行环境里，不管是“指向 int 的指针”，还是“指向 double 的指针”，在运行时都是相同的事物。可是，通过在 int 类型的变量之前加上&来取得它的指针，随后利用指针间接取出来的值，不出意外肯定是 int 类型。为什么？因为 int 和 double 的内部表示完全不同。

因此，如今的运行环境，像下面这样取得指向 double 类型变量的指针，之后将其赋给指向 int 的指针变量，编译器必定会提示警告。

```
int    *int_p;
double double_variable;

/*将指向 double 变量的指针赋予指向 int 的指针变量（恶搞！） */
int_p = &double_variable;
```

顺便说一下，在我的环境里出现了下面的警告：

```
warning: assignment from incompatible pointer type
```

下面的“指针运算”这一小节，会进一步说明“编译器会帮我们记住指针指向什么样的类型”的重要意义。

1.2.4　指针运算

C 语言的指针运算功能是其他语言所没有的。

指针运算是针对指针进行整数加减运算，以及指针之间进行减法运算的功能。

我们先来看一看下面这个例程（参照代码清单 1-2）。

【注意 1】

严格地说，代码清单 1-2 的程序并不符合 C 语言标准。

对于指针加减运算，标准只允许指针指向数组内的元素，或者超过数组长度的下一个元素。指针运算的结果也只是允许指针指向数组内的元素，以及超过数组长度的下一个元素（关于这一点，请参照 4.3.2 节的补充内容“指针可以指向数组最后元素的下一个元素”）。标准没有对除此之外的情况做出任何定义。在下面的例程中，因为对不是指向数组的指针 hoge_p 进行了加法运算，所以它在这一点上违反了 C 语言标准*。

在大多数的环境下，这个程序是可以运行的。为了有效地说明后面的问题，比起严格遵守标准，我还是选择了这个简单、直接的例程。

* 标准写道：“一个指向非数组对象的指针，和指向只包含一个元素（类型和前者相同）的数组的第一个元素的指针，具有相同的意义”。因此，只要你不做加 2 以上（包括 2）的加法运算就不会出现错误。

代码清单 1-2　pointer_calc.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int hoge;
 6:      int *hoge_p;
 7:
 8:      /*将指向 hoge 的指针赋予 hoge_p */
 9:      hoge_p = &hoge;
10:      /*输出 hoge_p 的值*/
11:      printf("hoge_p..%p\n", hoge_p);
12:      /*给 hoge_p 加 1*/
13:      hoge_p++;
14:      /*输出 hoge_p 的值*/
15:      printf("hoge_p..%p\n", hoge_p);
16:      /*输出 hoge_p 加 3 后的值*/
17:      printf("hoge_p..%p\n", hoge_p + 3);
18:
19:      return 0;
20:  }
```

我的环境中的结果如下：

```
hoge_p..0xbfbfd9e4  ←— 最初的值
hoge_p..0xbfbfd9e8  ←— 加 1 后的值
hoge_p..0xbfbfd9f4  ←— 加 1 之后再加 3 的值
```

第 9 行，将指向 hoge 的指针赋予 hoge_p，第 11 行输出 hoge_p 的值。我的环境里，hoge 被保存在门牌号为 0xbfbfd9e4 的地址中。

在第 13 行，使用运算符++，给 hoge_p 加 1。

输出结果……0xbfbfd9e4 变成了 0xbfbfd9e8，为什么不是增加了 1，而是增加了 4 呢？

在第 17 行，给加 1 后的 hoge_p 再加上 3，输出的结果由 0xbfbfd9e8 变成了 0xbfbfd9f4，增加了 12。

这就是指针运算的特征。在 C 语言中，对指针进行加 1 运算，地址的值会**增加当前指针所指向数据类型的长度**。例程中的 hoge_p 是“指向 int 的指针”，因为我的环境中 int 类型的长度为 4，所以给地址加 1，指针前进 4 个字节，给地址加 3，指针就前进 12 个字节。

> 要　点
> **对指针加 N，指针前进“当前指针指向的数据类型的长度×N”。**

【常见疑问之 3】
指针就是地址吧，给指针加 1，指针难道不应该前进 1 个字节吗？

这是最常见的疑问了。理解这一点的前提，需要先弄清楚 C 语言中指针和数组之间有什么样的微妙关系，以及为什么 C 中会存在指针运算这样**奇怪的**功能。

关于这些问题，稍后会进行说明，目前还是让我们带着疑问往下走吧。

1.2.5　什么是空指针

空指针是一个特殊的指针值。

空指针是指可以确保没有指向任何一个对象的指针。通常使用宏定义 NULL 来表示空指针常量值。

空指针确保它和任何非空指针进行比较都不会相等，因此经常作为函数发生异常时的返回值使用。另外，对于第 5 章的链表来说，也经常在数据的末尾放上一个空指针来提示：“请注意，后面已经没有元素了哦。”

在如今的操作系统下，应用程序一旦试图通过空指针引用对象，就会马上招致一个异常并且当前应用程序会被操作系统强制终止*。因此，如果每次都使用 NULL 来初始化指针变量，在错误地使用了无效（未初始化）的指针时，我们就可以马上发现潜在的 bug。

* 并不是所有的操作系统都能对空指针引用进行错误处理的。像 DOS 这样没有内存保护功能的操作系统也就罢了，连 UNIX 居然也允许通过空指针引用对象。

通常，我们可以根据指针指向的数据类型来明确地区别指针的类型。如果将“指向 int 的指针”赋给“指向 double 的指针”，如今的编译器会报出前面提到的警告。但是，只有 NULL，无论对方指向什么类型的变量，都可以被赋值和比较。

偶尔会见到先将空指针强制转型，然后进行赋值、比较操作的程序，这不但是**徒劳的**，甚至还会让程序变得难以阅读。



补充　NULL、0 和'\0'

经常有一种**错误的**程序写法：使用 NULL 来结束字符串。

```
/*
 *通常，C 的字符串使用'\0'结尾，可是因为 strncpy()函数在 src 的长度大于 len
 *的情况下没有使用'\0'来结束，所以一板一眼地写了一个整理成 C 的字符串形式的
 *函数（企图）
 */
void my_strncpy(char *dest, char *src, int len) {
    strncpy(dest, src, len);
    dest[len] = NULL;    ←—使用 NULL 来结束字符串！！
}
```

上面的代码，尽管在某些运行环境下能跑起来，但无论怎样它就是错误的。因为字符串是使用“空字符”来结束的，而不是用空指针来结束。

在 C 语言标准中，空字符的定义为“所有的位为 0 的字节称为**空字符**（null character）”（5.2.1）。也就是说，空字符是值为 0 的字符。

空字符在表现上通常使用'\0'。因为'\0'是常量，所以实际上它等同于 0。也许有些吓到你了，'\0'呀'a'呀什么的，它们的数据类型其实并不是 char，而是 int*。

* 如果是 C++，就不是这个结论了。

另外，在我的环境中，NULL 在 stdio.h 里的定义如下：

```
#define NULL    0
```

看到这个，你可能会说：“说来说去，那还不都是 0 嘛。”确实在大部分的情况下是这样的，但背后的事情却异常复杂。

正如前面说的那样，写成'\0'和写成常量的 0 其实是一样的。使用'\0'只不过是习惯使然。如果想让代码容易读，遵从习惯是非常重要的。

将 0 当作空指针来使用，除了极其例外的情况，通常是不会发生错误的。

但是，如果在字符串的最后使用 NULL，就必然会发生错误。

标准允许将 NULL 定义成(void*)0，所以在 NULL 被定义成(void*)的时候，如果使用 NULL 来结束字符串，编译器必然会提示警告。

看到刚才的关于 NULL 的定义，可能有人会产生下面的推测：

> 啥呀？所谓空指针，不就是为 0 的地址嘛。
>
> 在 C 中，为 0 的地址上应该是不能保存有效数据的吧？放什么都起不到任何作用，这没什么大不了的。

这种推测好像颇有道理，但也是有问题的。

确实在大多数的环境中，空指针就是为 0 的地址。但是，由于硬件状况等原因，世上也存在值不为 0 的空指针。

偶尔会有人在获得一个结构体之后，先使用 memset()将它的内存区域清零然后再使用。此外，虽然 C 语言提供了动态内存分配函数 malloc()和 calloc()，但是抱着“清零后比较好”的观点，偏爱 calloc()的人倒有很多。这样也许可以避免一些难以再现的 bug。

使用 memset()和 calloc()将内存区域清零，其实就是单纯地使用 0 来填充位。通过这种处理，当结构体的成员中包含指针的时候，这个指针能不能作为空指针来使用，最终是由运行环境来决定的。

顺便说一下，对于浮点数，即使它的位模式为 0，值也不一定为 0*。

* 整数类型还好，但是我还是感觉依赖环境编出来的代码是不干净的。

说到这里，

> 哦，原来这样啊，所以要使用宏定义的 NULL 呢。对于空指针的值不为 0 的运行环境，NULL 的值应该被#define 成别的值吧。

可能会有人产生以上的想法。实际上，这种想法也是**有偏差**的，这涉及问题的内部根源。

比如，尝试编译下面的代码：

```
int *p = 3;
```

在我的环境里，会出现以下警告：

```
warning: initialization makes pointer from integer without a cast
```

因为 3 无论怎么说都是 int 型，指针和 int 型是不一样的，所以编译器会提示警告。尽管在我的环境里指针和 int 的长度都是 4 个字节，但还是出现了警告。如今的编译器，几乎都是这样的。

继续，让我们尝试编译下面的代码：

```
int *p = 0;
```

这一次没有警告。

如果说将 int 型的值赋予指针就会得到一个警告，那么为什么值为 3 的时候出现警告，值为 0 的时候却没有警告呢？简直匪夷所思！

这是因为在 C 语言中，“当常量 0 处于应该作为指针使用的上下文中时，它就作为空指针使用”。上面的例子中，因为接受赋值的对象为指针，**编译器根据上下文判断出**“0 应该作为指针使用”，所以将常数 0 作为空指针来读取。

无论如何，编译器都会针对性地对待“需要将 0 作为指针进行处理的上下文”，所以即便是空指针的值不为 0 的情况下，使用常量 0 来代替空指针也是合法的。

此外，如上所述，有的环境中像下面这样定义 NULL：

```
#define NULL ((void*)0)
```

ANSI C 中，根据“应该将 0 作为指针进行处理的上下文”的原则，将常量 0 作为指针来处理。因此，显式将 0 强制转型成 void*是没有意义的。但是在某些情况下，编译器也可能会理解不了“应该将 0 作为指针进行处理的上下文”。

这些情况是：

- 没有原型声明的函数的参数
- 可变长参数函数中的可变部分的参数

ANSI C 中，因为引入了原型声明，只有在你确实做了原型声明的情况下，编译器才能知道你“想要传递指针”。

可是，对于以 printf()为代表的可变长参数函数，其可变部分的参数的类型编译器是不能理解的。另外糟糕的是，在可变长参数的函数中，还经常使用常量 NULL 来表示参数的结束（比如 UNIX 的系统调用 execl()函数）。

以上情况下，简单地传递常量 0，会降低程序的可移植性。

因此，通过使用宏定义 NULL 来将 0 强制转型成 void*，可以显式地告之编译器当前的 0 为指针*。

* 关于这个话题，在 C 语言 FAQ（http://www.catnet.ne.jp/kouno/c_faq/c_faq.htm）中，也花费了一章的笔墨进行了讨论。

1.2.6 实践——swap 函数

到这里为止，已经对指针进行了大致的介绍，但是关于指针的用处还没有解释。

在这里，我们使用经常用于展示指针使用方法的例程——招牌的 swap 函数来进行下面的说明。

下面这个函数试图交换两个 int 类型变量的值，虽然这个例子总让人觉得不太自然，但我们这里还是使用了这个例子。

```
void swap(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

让我们调用一下这个函数。

```
int x, y;

x = 5;
y = 10;

swap(x, y);

printf("x..%d y..%d\n", x, y);
```

通过运行以上的调用，我们发现 x 和 y 的值并未交换。

调用 C 的函数，参数传递往往是**传值**，这种方式传递的是参数的副本。

可能会有人这样想：

> 啥？这本书也谈传值的问题？以前俺买的那些 C 语言入门书中也有这个内容呢。也罢，姑且先听你说说看，所谓的传值究竟是怎么回事？

为了这部分读者，我换个角度来说明一下。

这个例子中的 swap 函数，有两个 int 型的参数。所以，也一定可以通过下面的方式调用这个函数：

```
swap(3, 5);
```

那么，在 swap 这一边，我们先排除形参（这里是 a 和 b）在调用的时候被设定值的情况，将它们和通常的局部变量同样对待。当然，赋值也是可以的。

假设给 a 和 b 赋值会让调用方的变量给 x、y 带来影响，那么像 swap(3,5)这种方式的调用，究竟会发生什么呢？常量 3 变成 5，5 变成 3？绝不可能。

顺便提一下，有一些 C 语言之外的其他语言，给函数的形参赋值是会影响到调用方的实参的。在以前的 FORTRAN 中，所有的参数都是这样的。在将常量作为参数进行传递的时候，稀里糊涂地给形参赋了值——结果有可能惨不忍睹。在 Pascal 中，为了不给调用方的变量带来影响，在定义函数的时候，特别地指定参数为**变量参数**。如果试图给变量参数指定常量，编译器会报错。

C 语言里完全没有这样的现象。无论如何，**函数的形参都和调用时被设定值的局部变量一样**。如果不是这样的话，就会背负 FORTRAN 那样的危险，或者像 Pascal 那样在语法下做些文章。从 C 语言的发展过程来看，采取 Pascal 那样麻烦的方式是不可能的。

因此，在 C 里面想要改写调用方的变量，可采取传递指针的方式。

```
void swap(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

调用方式为：

```
swap(&x, &y);
```

在本例中，向函数传递指向 x 和 y 的指针（也就是地址）。尽管指针是通过传值的方式进行传递的，但由于在 swap 中使用了*运算符，所以通过指针可以间接访问到 x 和 y。向 swap 传递的是地址，x 和 y 自身并没有移动。

给大家举一个例子。有一位从不信任部下，甚至神经质得**有点让人讨厌的**上司，每当他向部下分配任务的时候，总是将复制后的文档交给部下。这

些部下无论多么地努力，也不可能调换这位上司的文件柜中存在的“文档 A”和“文档 B”的内容。除非上司这样吩咐他的部下：“给我将书柜某个地方的文档 A，以及书柜某个地方的文档 B 调换一下！”如果告知了文档的“地点”，这些部下就能调换文档内容了。就是这么一回事。

如果换个方式说明这一小节开头的那个例子，就好像冷不丁地向 swap 函数要求 “帮我把 5 和 10 交换过来”。

换成其他的函数，

```
a = 5;
func(a);
```

或

```
func(5);
```

你不认为它们是一样的吗？

后面的那个例子向 swap 函数提出了“请交换这里的变量和那里的变量”的要求。显然这个要求是可以满足的。

说个题外话，如果仅仅是需要交换整型变量的值，完全不使用临时变量也是可以的。比如使用下面的宏定义：

```
#define SWAP(a, b) (a += b, b = a - b, a -= b)
```

在这种方式（还可以使用异或运算符）下，在颠倒使用同一个变量时，这个程序是不能正常运行的。比如你写了 SWAP(a[i],a[j])，并且恰巧 i == j，那我只能恭喜你中招了。当然，如果你能担保这种情况永远不可能出现，使用这个宏也未尝不可。

如果到现在为止，对以上内容还是不太明白，请阅读第 2 章。第 2 章会具体讲解当提到形参是实参的副本时，实参究竟被复制到哪里，以及怎样实现复制。



补充　**形参和实参**

几乎所有的 C 语言的入门书籍中，都会讲解“形参”和“实参”的概念。但是它们还是经常被轻易混淆。

实参是调用函数时的参数。

```
func(5);    ←—这里的 5 是实参。
```

形参是接受实参的一方。

```
void func(int hoge)    ←—这里的 hoge 是形参
{
    ⋮
}
```

后面会经常出现“形参”、“实参”这样的词，请大家一定注意不要混淆它们。

1.3　关于数组

1.3.1　运用数组

数组是指将固定个数、相同类型的变量排列起来的对象。

还是先让我们来体验一把（参照代码清单 1-3）。

运行结果如下：

```
0
1
2
3
4
&array[0]... 0xbfbfd9d4
&array[1]... 0xbfbfd9d8
&array[2]... 0xbfbfd9dc
&array[3]... 0xbfbfd9e0
&array[4]... 0xbfbfd9e4
```

代码清单 1-3　array.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int array[5];
 6:      int i;
 7:
```

```
 8:     /*为数组array的各元素设值*/
 9:     for (i = 0; i < 5; i++) {
10:         array[i] = i;
11:     }
12:
13:     /*输出数组各元素的值*/
14:     for (i = 0; i < 5; i++) {
15:         printf("%d\n", array[i]);
16:     }
17:
18:     /*输出数组各元素的地址*/
19:     for (i = 0; i < 5; i++) {
20:         printf("&array[%d]... %p\n", i, &array[i]);
21:     }
22:
23:     return 0;
24: }
```

在第 5 行，声明数组类型变量 array。

在第 9～11 行，为 array 的各元素设定值。这里就是单纯地依次将 0 赋给 array[0]，将 1 赋给 array[1] ……

在第 14～16 行，输出数组各元素的值，就是运行结果最前面的那 5 行。

在第 19～21 行，输出数组各元素的地址。观察输出的地址，可以发现这些地址的值与值之间都相差 4 个字节。

我的环境中，int 的长度正好是 4 个字节，内存中数组的布局如图 1-4 所示。

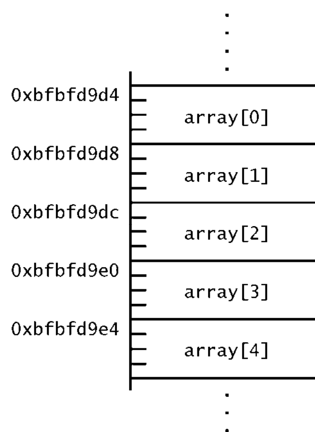


图 1-4　数组在内存中的布局

在本章 1.2.4 节中，曾经提到“对指针加 N，指针前进‘当前指针指向的数据类型的长度 × N’”的原则。在这里将会重新提起这个话题，更详细的说明请阅读下一节。



补充

C 的数组是从 0 开始的

使用 C 语言，声明一个数组变量：

```
int hoge[10];
```

这里指定的 10 是数组元素的个数，因为在 C 中，数组的下标从 0 开始，通过上面这个声明，你可以使用 hoge[0]～hoge[9]，但你不能使用 hoge[10]。

这个规则经常会让菜鸟们犯迷糊。

FORTRAN 的数组就是从 1 开始的，要是 C 跟 FORTRAN 一样那该多好……有读者会这么想吧？

这样真的好吗？我觉得你需要三思而行。

打个比方，我上班的公司位于名古屋的一座 5 层的写字楼里，假设某人每爬一层楼花费 10 秒钟，那么此人如果从地面爬上 5 楼，一共需要多少秒呢？50 秒？恭喜你，**答错了**，答案是 40 秒。

相信大家在中学里都学过“等差数列”，等差数列第 n 项等于“初项 + 公差 × $(n\text{-}1)$”。

“1900 年代”[①]不是 19 世纪，它的一大半属于 20 世纪。更让人纠结的是，2000 年不属于 21 世纪，而属于 20 世纪。

对于这些现象，如果把

- 写字楼和地面相同高度的那层，计数为 0 层，
- 数列最初的项，计数为 0 项，
- 最初的世纪计数为 0 世纪，公历最初的年计数为 0 年，

就能够回避问题。

平时编程中，也经常发生“差 1 错误”问题。普遍的观点是使用 0 作为基准进行编号。

如果还是有人不太理解，可以再举一个和编程相关的例子。

C 语言中可以使用二维数组（准确地说应该是“数组的数组”），但必须在编译的时候知道二维数组的宽度。

---

① “××××年代”是日本的年代表达方式，“1900 年代”指 1900 ~ 1999 年。——译者注

假设我们执意要用一维数组去代替宽度可变的二维数组使用，

```
/* width 为行的宽度，引用第 line 行，第 col 列的元素*/
array[line * width + col]
```

假设最初的行为第 1 行，最初的列为第 1 列，并且数组的下标是从 1 开始，就需要把上面的代码修改成下面这样，

```
array[(line-1) * width + (col-1)]
```

C 的数组下标从 0 开始，还有一个语法上的原因（后面会提到）。

如果你使用习惯了，从 0 开始的数组比起从 1 开始的数组，使用起来方便得多。

> 反正如今的内存已经很大了，还不如声明数组的时候多一个元素的长度，下标就可以从 1 开始使用了。

比起这种**敷衍了事的想法**，我们还不如让自己习惯于从 0 开始使用数组，除非你正在做 FORTRAN 程序的移植工作。

1.3.2 数组和指针的微妙关系

正如之前说明的那样，给指针加 N，指针前进“当前指针指向的变量类型的长度 × N”。

因此，给指向数组的某个元素的指针加 N 后，指针会指向 N 个之后的元素。

代码清单 1-4 array2.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int array[5];
 6:      int *p;
 7:      int i;
 8:
 9:      /*给数组 array 的各元素设定值*/
10:      for (i = 0; i < 5; i++) {
11:          array[i] = i;
12:      }
13:
```

```
14:        /*输出数组各元素的值（指针版）*/
15:        for (p = &array[0]; p != &array[5]; p++) {
16:            printf("%d\n", *p);
17:        }
18:
19:        return 0;
20:    }
```

运行结果如下。可以发现运行结果和代码清单 1-3 的前半部分相同。

```
0
1
2
3
4
```

从第 15 行开始一个 for 循环，最初指针 p 指向 array[0]，通过 p++顺序地移动指针，引导指针指向 array[5]（尽管它不存在）（请参照图 1-5）。

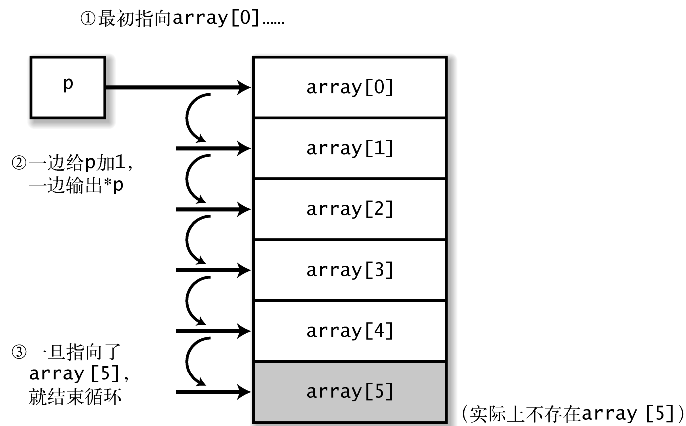


图 1-5　利用指针输出数组的值

使用++运算符给指针加 1，指针前进 sizeof(int)个字节。

此外，第 15 ~ 17 行的代码也可以换一种写法（我们可以称之为“改写版”）。

```
/*利用指针输出数组各元素的值——改写版*/
p = &array[0];
for (i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));
}
```

这种写法里，指针并没有一步步前进，而是固定的，只是在打印的时

候加 i。

话说回来，你觉得这种写法容易阅读吗？

至少在我看来，无论写成 p++，还是*(p + i)，都不容易阅读。还是最初的例子中 a[i]这样的方式更容易理解。

实际上，本书主张的是“因为利用指针运算的写法不容易阅读，所以让我们**抛弃这种写法吧**”。

先把写法好坏的问题放在一边。事实上，指针运算是 C 语言的一个“**奇怪**”的功能。到底有多“**奇怪**”，片刻之后就为你一一道来。

1.3.3 下标运算符[]和数组是没有关系的

在前一小节的“改写版”例程中，像下面这样将指针指向数组的初始元素。

```
p = &array[0];
```

其实也可以写成下面这样：

```
p = array;
```

对于这种写法，很多 C 语言的入门书籍是这样说明的：

> 在 C 中，如果在数组名后不加[]，单独地只写数组名，那么此名称就表示“指向数组初始元素的指针”。

在这里，我可以负责地告诉你，上面的说明是错误的*。

* 如果考虑给人留点面子，其实我应该这么说：“不能说这个说明总是对的。”可是考虑一下听到这个说明的人如何解释它，就感觉还不如痛痛快快地指出来“这个说明完全是错误的”。

又惊着你了吧？

在 C 的世界里，事到如今你再去否定“数组名后不加[]，就代表指向初始元素的指针”这个“强大的”**误解**显得有点无奈。对于这种已经深入人心的观点，你突然放言它其实是个误解，可能很多人无法接受。下面让我们依法来证明。

将&array[0]改写成 array，“改写版”的程序甚至可以写成下面这样：

```
p = array;     ←只是改写了这里，可是……
for (i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));
}
```

另外，程序中*(p + i)也可以写成 p[i]。

```
p = array;
for (i = 0; i < 5; i++) {
    printf("%d\n", p[i]);
}
```

也就是说，

```
*(p + i)
```

和

```
p[i]
```

是同样的意思。可以认为后面的写法是前面的简便写法。

在这个例子中，最初通过 p = array;完成了向 p 的赋值，但之后 p 一直没有发生更改。所以，早知如此，何必当初偏要多声明一个 p，还不如一开始就写成 array 呢。

```
for (i = 0; i < 5; i++) {
    printf("%d\n", array[i]);
}
```

呀，好像又回去了呢。

结论就是，

```
p[i]
```

这种写法只不过是

```
*(p + i)
```

这种写法的简便写法，**除此之外，它毫无意义**。array[i]和 p[i]有什么不一样吗？array[i]也可以像 p[i]一样，将 array 解读成“指向数组的初始元素的指针”。

也就是说，存在

```
int array[5];
```

这样的声明的时候，“一旦后面不追加[]，只写 array”并不代表要使 array 具有指向数组第 1 个元素的指针的含义，无论加不加[]，**在表达式中，数组都可以被解读成指针**。

顺便说一下，对于这个规则来说，有三个小的例外，我们会在第 3 章作详细说明。

你可以认为这是一个哗众取宠的异端邪说，但至少在语法上，数组**下标**

运算符[]和数组无关。

这里也是 C 的数组下标从 0 开始的理由之一。

> 要 点
> **【非常重要!!】**
> **表达式中，数组可以解读成“指向它的初始元素的指针”。尽管有三个小例外，但是这和在后面加不加[]没有关系。**

> 要 点
> **p[i]是*(p + i)的简便写法。**
> **下标运算符[]原本只有这种用法，它和数组无关。**

需要强调的是，认为[]和数组没有关系，这里的[]是指在表达式中出现的下标运算符[]。

声明中的[]，还是表达数组的意思。也就是说，声明中的[]和表达式中的[]**意义完全不同**。表达式中的*和声明中的*的意义也是完全不同的。这些现象使得 C 语言的声明在理解上变得更加扑朔迷离……对此，第 3 章将会进行详细的说明。

此外，如果将 a + b 改写成 b + a，表达式的意义没有发生改变，所以你可以将*(p + i)写成*(i + p)。其次，因为 p[i]是*(p + i)的简便写法，实际上它也可以写成 i[p]。

引用数组元素的时候，通常我们使用 array[5]这样的写法。其实，就算你写成 5[array]，还是可以正确地引用到你想要的元素。可是，这种写法实在太另类了，它不能给我们带来任何好处。

> 要 点
> **p[i]可以写成 i[p]。**

> 要 点
> **【比上面这个要点更重要的要点】**
> **但是别写成那样。**



语法糖

p[i]是*(p+i)的简单写法，实际上，至少对于编译器来说，[]这样的运算符完全可以不存在。

可是，对于人类来说，*(p + i)这种写法在解读上比较困难，写起来也麻烦（键入量大）。因此，C 语言引入了[]运算符。

就像这样，这些仅仅是为了让人类容易理解而引入的功能，的确可以让我们感受到编程语言的甜蜜味道（容易着手），有时我们称这些功能为语法糖（syntax sugar 或者 syntactic sugar）。

1.3.4　为什么存在奇怪的指针运算

如果试图访问数组的内容，老老实实地使用下标就可以了。为什么存在指针运算这样奇怪的功能呢？

其中的一个原因就是受到了 C 的祖先 B 语言的影响。

在 1.1.2 节的补充内容中也提到了，B 是一种“没有类型”的语言。B 中可以使用的类型只有 word 型（也就是整型），指针也是作为整型来使用的（像浮点型这样高级的事物，你根本见不到）。B 是虚拟机上运行的解释器，这个虚拟机以 word 为单位分配内存地址（如今普通的计算机以字节为单位）。

由于 B 以 word 为单位，如果指针（仅仅是表现地址的简单的整数）加 1，指针就指向数组的下一个元素。为了继承这种特性，C 引入了“指针加 1，指针前进它所指向类型的长度”这个规则*。

* 关于这方面的论述在“The Development of the C Language”[5]这篇论文（也可以认为是随笔吧）中有记载。你可以从 Dennis Ritchie 的网站上获取它。

B 语言中同样存在 p[i]是*(p + i)的语法糖这样的规则。可是，这里的(p + i)只不过是单纯的整数之间的加法运算*。

* 因此，在 B 中，将 p[i]写成 i[p]是一件“理所当然”的事。但是这种规则居然被原封不动地搬到了 C 中（有点悲哀！）。

解引用*、地址运算符&，也以几乎和 C 相同的形态存在于 B 语言中。

另外还有一个理由就是，早先使用指针运算可以写出高效的程序。

通常情况下，我们总是使用循环语句来处理数组，一般都写成下面的}形式，

```
for (i = 0; i < LOOP_MAX; i++) {
    /*
     *在这里，使用 array[i]进行各种各样的处理。
```

```
        * array[i]会出现多次。
        */
}
```

array[i]在循环中会出现多次，每次都要进行相当于*(array + i)的加法运算，效率自然比较低。

因此，可以使用指针运算重写上面这段循环，

```
for (p = &array[0]; p != &array[LOOP_MAX]; p++) {
    /*
     *在这里，使用*p 进行各种各样的处理。
     * *p 会出现多次。
     */
}
```

尽管*p 在循环内部会出现多次，但加法运算只有在循环结束的时候执行一次。

K&R p.119 中叙述了“一般情况下，使用指针的程序比较高效”。上面的说明应该可以作为这段叙述的根据吧。

可是，这些**无论怎样都是老黄历**了。

如今，编译器在不断地被优化，对于循环内部重复出现的表达式的集中处理，是编译器优化的基本内容。对于现在一般的 C 编译器，无论你使用数组还是指针，效率上都不会出现明显的差距。基本上都是输出完全相同的机器码。

总的来说，C 的指针运算功能的出现，**源自于早期的 C 自身没有优化手段**。这一点并不奇怪，请大家回想一下在前面介绍过的内容，C 本来只是为了解决开发现场的人们眼前的问题而出现的一种语言。Unix 之前的 OS 几乎都是使用汇编写的，即使晦涩难懂，人们也不会大惊小怪。对于当时的环境，追求什么编译器优化实在有点勉为其难。因此，**当初开发 C 语言的时候**，是完全有必要提供指针运算功能的。可是……

1.3.5 不要滥用指针运算

被称为 C 语言宝典的 *K&R* 指出：“一般情况下，使用指针的程序比较高效。”这完全是“**那个时代的错误**”。

可是，正如前面所说，对于如今的编译器，无论是使用指针运算还是下标运算，都生成几乎完全相同的执行代码。

事到如今……难道不应该放弃使用指针运算*，老老实实地使用下标访问吗？

* 下标运算符也是“指针运算＋解引用运算符”，这里提到的“指针运算”是指明确地对指针进行加减运算的程序写法。

虽然 *K&R* 被很多人奉为“神书”，可是对于我来说，它连作为菜鸟实习的资料也不够格。为什么这么说？因为在此书中，那些滥用指针的例程完全可以让你崩溃。

莫名其妙地使用像*++args[0]这样的语句，并且乐此不疲，实在让人心烦。

K&R 里面记载了下面这个作为 strcpy()实现的例子：

```
/* strcpy: 将 t 复制到 s;指针版 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

> 虽然乍一看不容易理解，但是这种写法其实是非常方便的。因为会在 C 程序中经常遇到，所以我们应该掌握这种惯用写法。

既然知道“乍一看不容易理解”，那就不应该这样写，难道不是吗？*

* 特别是，在这段代码中，当循环结束后，指针指向了空字符的下一个字符，之后如果继续复制其他字符串，会很容易诱发 bug。

满大街的 C 语言入门书都在教育我们，使用指针运算比使用下标会让程序

- 更有效率
- 更有 C 语言范儿

所谓的“更有效率”，只不过是臆想罢了。对于这种“微不足道的”优化工作，与其让人去小心翼翼地做，还不如交给编译器来干。

所谓“更有 C 语言范儿”好像是有些道理。如果只是为了要让程序“有范儿”，而让代码变得晦涩难懂，那么还是拜托你行行好，扔掉这种恶习吧。

在学校里，我们要完成一些课后作业。好不容易完成了一个使用下标的程序题，不料后面的那道题为“请使用指针将刚才那道题的程序重新完成一遍”。这种事常有吧。

老实说，这种事很无聊。也许你会很“威武”地依然使用下标原封不动地把程序又写了一遍，然后交给了老师。面对老师的指责，你义正辞严：

> 咦，下标运算符[]只不过是指针运算的语法糖而已，在本质上这样的写法也是在使用指针啊。

尽管这样，这位可爱的老师可能还是不会放过你，于是你就急了：

行，不就是把像 p[i]这样使用下标的地方，机械地一个个替换成*(p+i)嘛。

话说回来，丢了学分，我可不负责哟。在 C 的世界里，使用指针运算要比使用下标的写法让人感觉更“帅一些”。

但是……与其在这些无聊的地方“耍酷”，倒不如多花点时间学一些有用的知识。你要知道，作为一个程序员，还有堆积如山的知识等着你去掌握呢。

当然，什么样的规则都有例外，比如，在“一个巨大的 char 数组中，参杂了各种类型的数据[①]，并且我们试图读取第多少字节的数据”这样的情况下，还是使用指针运算写的程序比较容易理解。

此外，作为一个 C 程序员连指针运算的代码也读不懂，多少有点可悲。

尽管如此，让我们至少从现在开始尽量使用下标来写新的程序，这样做对自己，以及对以后有机会阅读你的程序的人，都有好处。



补充　修改参数，好吗?

刚才那个在 *K&R* 中记载的实现 strcpy()的例子中，使用++直接修改了形参 s 和 t 的值。

确实，C 的形参可以和事先被设定值的局部变量同样使用，对值进行修改在语法上并没有任何问题。但我从来不这么做。

函数的参数是从调用方得到的非常重要的信息，如果一时疏忽错误地修改了参数，就再也恢复不了了。对于在后面追加新的逻辑，或者调试程序的情况下，因为原始的参数已经被修改，如果想要看一下参数的值，你会感觉非常棘手。

此外，参数都应该有一个有意义的名称（刚才的 strcpy()是个反面教材）。在修改参数的时候，违背最初参数名称的意义的“恶行”*也屡见不鲜。

顺便说一下，Ada 和 Eiffel 不允许修改作为输入信息的函数参数*。

* 这种恶行多发生在循环计数的逻辑中。
* 在内部，我认为应该是和 C 采用了大抵相同的参数传递方式。

1.3.6　试图将数组作为函数的参数进行传递

在这里，让我们首先来做一个具有实用价值的例子：从英文的文本文件

① 在网络通讯中，数据交换程序中经常会使用一个 char 数组保存各种数据类型的数据。我们通常将这种行为称为序列化。——译者注

中将单词一个一个取出来。

关于调用方式，模仿 fgets()，定义成下面的形式：

```
int get_word(char *buf, int buf_size, FILE *fp);
```

函数的返回值是单词的字母个数，当读到文件末尾的时候返回 EOF。

对于单词的定义，如果仔细考虑一下，好像还真不是件容易的事。这里我们选择使用 C 的 isalnum()这个宏（ctype.h）。如果返回真，就表示是连续的几个字符那就作为单词，否则就是空白字符。

单词长度大于 buf_size 的情况下，因为处理会变得比较麻烦，我们考虑使用 exit()果断地结束程序。

为了测试这个函数，在程序中添加 main()用来驱动测试过程（调用 get_word()）（参照代码清单 1-5）。

main()中声明的数组 buf，在 get_word()中被填充值。

在 main()中，buf 作为函数的参数传递，因为这里是在表达式中，所以 buf 可以解读成"指向数组初始元素的指针"。因此，接受 buf 的 get_word()才可以像

```
int get_word(char *buf, int buf_size, FILE *fp)
```

这样，合法地接受 char *。

其次，在 get_word()中，可以像 buf[len]这样操作 buf 的内容。那是因为 buf[len]是*(buf + len)的语法糖。

一旦在 get_word 中使用下标运算符访问 buf 的内容，倒还真的会让人感觉从 main()传递过来的是 buf 这样的数组。显然这是个错觉，无论如何，从 main()传递过来的是指向 buf 的初始元素的指针（请回忆一下我们曾经提到的"C 是本来只能使用标量的语言"这个观点，参照 1.1.8 节）。

代码清单 1-5　get_word.c

```
1:  #include <stdio.h>
2:  #include <ctype.h>
3:  #include <stdlib.h>
4:
5:  int get_word(char *buf, int buf_size, FILE *fp)
6:  {
7:      int len;
```

```
 8:     int ch;
 9:
10:     /*跳过读取空白字符*/
11:     while ((ch = getc(fp)) != EOF && !isalnum(ch))
12:         ;
13:     if (ch == EOF)
14:         return EOF;
15:
16:     /*此时，ch 中保存了单词的初始字符。*/
17:     len = 0;
18:     do {
19:         buf[len] = ch;
20:         len++;
21:         if (len >= buf_size) {
22:             /*由于单词太长，提示错误*/
23:             fprintf(stderr, "word too long.\n");
24:             exit(1);
25:         }
26:     } while ((ch = getc(fp)) != EOF && isalnum(ch));
27:     buf[len] = '\0';
28:
29:     return len;
30: }
31:
32: int main(void)
33: {
34:     char buf[256];
35:
36:     while (get_word(buf, 256, stdin) != EOF) {
37:         printf("<<%s>>\n", buf);
38:     }
39:
40:     return 0;
41: }
```

准确地说，在 C 中是**不能**将数组作为函数参数进行传递的。但是，你可以通过传递指向初始元素的指针来达到将数组作为参数进行传递的目的。

> 要　点
> **如果试图将数组作为函数参数进行传递，那就传递指向初始元素的指针。**

可是，一般情况下将 `int` 等作为参数进行传递的时候，与在当前的例子中将数组作为参数进行传递的时候，它们的传递方式是完全不同的。

在 C 中，函数参数传递都是传值，向函数传递的都是参数的副本。当前的例子同样如此，向 `get_word()`传递的是指向 `buf` 初始元素的指针的副本。但是，`main()`和 `get_word()`引用的都是 `buf` 本身，而不是 `buf` 的副本。正因为如此，`get_word()`才能正确地向 `buf` 填充字符串的内容。



补充　**如果对数组进行值传递**

在迫不得已的情形下，如果你执意要将数组的副本作为参数进行传递，可以使用替代方法——将数组的所有元素整理成结构体的成员。

正如 1.1.8 节中说明的那样，C 本来就是只能使用标量的语言。但是这个问题在比较早的时期就得到了改善，我们现在已经可以对结构体进行赋值了。

可是，这种方法在效率上是有问题的，关于这一点你需要有心理准备。当在操作一个巨大的数组的时候，如果对所有元素一一复制，那可是非常耗时的。

我以前在模拟奥赛罗棋游戏[①]的行棋思路的时候，曾经通过这种方法来对表示盘面形势的二维数组进行值传递。在行棋思路中，通过对“如果这样下，形势就会这样”的巨大的树结构进行不断地递归探寻，来得到最优的出棋对策，因此，针对每一着棋都需要对记录盘面形势的二维数组进行复制。我感觉，也就是在这种需求下才会使用这种技术。

要　点

无论如何都要将数组进行值传递的时候，建议将数组整体整理成结构体成员。

1.3.7　声明函数形参的方法

本书的例程中，将 get_word()的参数像下面这样使用 char *进行声明，

```
int get_word(char *buf, int buf_size, FILE *fp)
```

“咦？俺可是一直这么写的哦”

```
int get_word(char buf[], int buf_size, FILE *fp)
```

应该有同学是这样想的吧。

只有在声明函数形参时，数组的声明才可以被解读成指针。

比如，对于

```
int func(int a[])
```

① 奥赛罗棋是一种双人棋盘游戏。在划分 64 格的棋盘上排列正反面为黑白色的圆形棋子，夹住对方棋子时可把它翻面换成己方的棋子的颜色，以此争胜负。——译者注

编译器可以针对性地解读成：

```
int func(int *a)
```

即使像下面这样定义了元素个数，编译器也是无视的，

```
int func(int a[10])
```

这也是语法糖之一。

必须要引起注意的是，int a[]和 int *a 具有相同意义，在 C 的语法中**只有这么一种情况**。关于这一点，在第 3 章中会有具体的说明。

> 要　点
>
> **在下面声明的形参，都具有相同的意义。**
>
> ```
> int func(int *a);      /*写法 1*/
> int func(int a[]);     /*写法 2*/
> int func(int a[10]);   /*写法 3*/
> ```
>
> **写法 2 和写法 3 是写法 1 的语法糖。**



补充　C 语言为什么不做数组下标越界检查?

通常，C 对数组的长度范围是不做检查的。“托它的福”，当向数组越界写入数据的时候，经常产生“**内存被破坏**”的问题。如果在较早的阶段，操作系统发现异常并且提示 Segmentation fault，或者“强制关闭异常的应用程序”这样的消息还算幸运。最不幸的是，相邻变量的值已经被破坏，程序却还在继续运行，并且你无法预知悲剧会在何时何地发生。

频繁地进行范围检查会影响效率，但至少应该让我们在编译的时候可以使用一个选项，以便要求编译器在调试模式下编译程序的时候，帮我们实施数组下标范围的检查。有这样的想法的人，不只是我一个吧。

但是，请稍微再想一想这个问题。

可以使用 int a[10];这样的方式声明数组，并且通过 a[i]的方式引用数组元素的那些编程语言，可以比较容易地进行数组长度范围检查。但是对于 C，当数组出现在表达式中的时候，它会立刻被解读成指针。此外，使用其他的指针变量也可以指向数组的任意元素，并且这个指针可以随意进行加减运算。

引用数组元素的时候，虽然你可以写成 a[i]，但是它只不过是*(a + i)的语法糖。

还有，当你向一个函数传递数组的时候，实际上你传递的是一个指向初始元素的指针。如果这个函数还存在于其他的代码文件中（另外一个编译单元），那么通过编译器是不可能追踪到数组的。

要求这样的语言在编译时生成检查数组长度的代码，是不是有些强人所难?

如果无论如何都需要进行数组长度检查，可以考虑将指针封装成结构体那样，运行时让指针自身持有可取值范围的信息。可是这么做对性能的影响很大，同时，也丧失了非调试模式下编译后的库和指针的兼容性。

总的来说，除了某些解释型的编程语言之外，目前几乎没有编译器可以为我们做数组的越界检查。

第 2 章

做个实验见分晓——C 是怎么使用内存的

2.1 虚拟地址

关于“地址”的概念，我们在第 1 章已经做了如下说明：

> 变量总是保存在内存的“某个地方”。“某个地方”这样的说法不容易理解，因此，就像使用“门牌号”确定“住址”一样，在内存中，我们给变量分配“门牌号”。在 C 的内存世界里，“门牌号”被称为“地址”。

经过以上的说明，有人可能会有这样的想法：

> 哦，原来是这样子的！俺的机器是 128MB 的内存，这 128MB 的内存是以字节为单位从 0 开始分配了连续的编号呀。也就是说，如果在俺的机器里用十进制来粗略地计数，从 0 开始有大约 128 000 000 个地址吧*。那么使用 `printf()`打印指针的值，其结果究竟是什么样的呢？

* 准确地说，应该是 128×1024×1024，结果是 134 217 728。

但是，如今的计算机可不是那么简单的哦。

现在的 PC 机和工作站的操作系统大多都提供了多任务环境，可以同时运行多个应用程序（进程）。那么，假设同时运行两个应用程序，然后尝试打印各自的变量地址，会出现一致的结果吗？若遵循刚才的推论，不会。

好吧，我们一起做个实验。首先，请将代码清单 2-1 编译成可执行的文件。

这里可执行文件的名称为 `vmtest`。

代码清单 2-1　vmtest.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int         hoge;
 6:      char        buf[256];
 7:
 8:      printf("&hoge...%p\n", &hoge);
 9:
10:      printf("Input initial value.\n");
11:      fgets(buf, sizeof(buf), stdin);
12:      sscanf(buf, "%d", &hoge);
13:
14:      for (;;) {
15:          printf("hoge..%d\n", hoge);
16:          /*
17:           * getchar()让控制台处于等待输入的状态.
18:           * 每次敲入回车键，增加 hoge 的值
19:           */
20:          getchar();
21:          hoge++;
22:      }
23:
24:      return 0;
25:  }
```

如今的操作系统，大多提供了多窗口环境，请你在自己的操作系统中打开**两个新的窗口**。若习惯用 UNIX，请使用 kterm（或者类似的工具）；若习惯用 Windows，请使用 DOS 窗口。另外，如果你**没有使用完全相同的方式启动**这些窗口，后面的实验可能会进行得不太顺利。Windows 的话，通过开始菜单打开两个新窗口就 OK。

然后，请在两个窗口分别试着运行刚才的程序（如果有必要，可以通过 `cd` 命令进入程序所在的目录）。

在我的环境里，得到了如图 2-1 所示的结果。

在第 8 行，通过 `printf()`输出变量 `hoge` 的地址，然后通过第 11 行的 `fgets()`，程序进入了等待输入的停止状态。显然，启动后的两个窗口肯定都处于运行状态，但 `hoge` 的地址却**两边完全相同**。

这两个进程的 `hoge` 看上去地址完全相同，但它们确实是在各自进程里面彼此独立无关的两个变量。根据 `Input initial value.`的屏幕窗口提示，我们接着输入一个任意值，这个值被赋予变量 `hoge`（第 12 行），在第 15 行又被 `printf()`输出。随后，`getchar()`让程序处于等待输入状态。每次敲击回

车键，hoge 的值都会增长并且被输出到窗口。正如我们看到的那样，这两个 hoge 内存地址明明是相同的，却各自保持着不同的值。

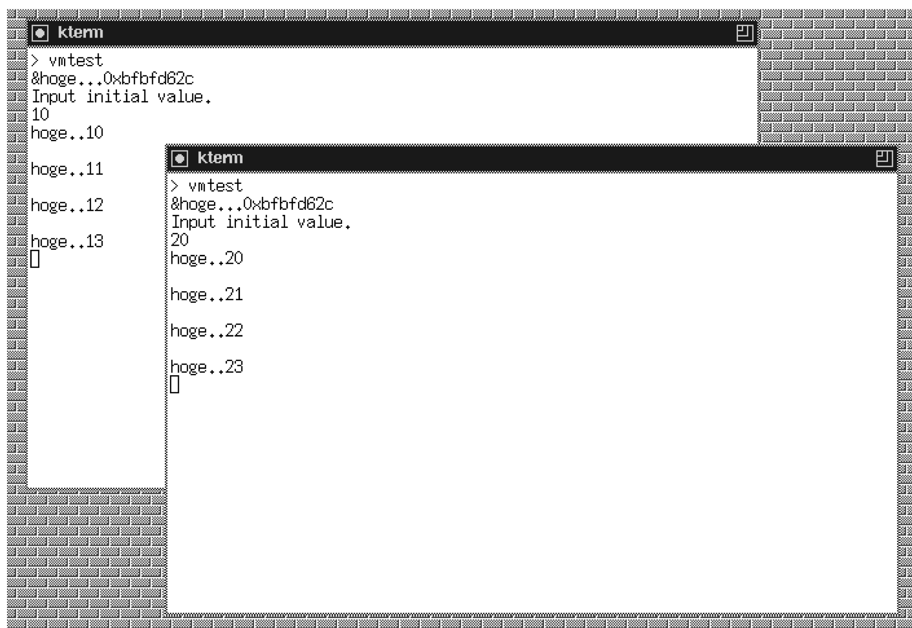


图 2-1 通过两个进程同时表示变量的地址

我在 FreeBSD 3.2-RELEASE 和 Windows 98 上运行了这个实验，都得到了以上的结果（编译器是 gcc）。

通对这样的实验发现，在如今的运行环境中，使用 printf()输出指针的时候，打印输出的并不是物理内存地址本身。

当今的操作系统都会给应用程序的**每一个进程分配独立的**“虚拟地址空间”。这和 C 语言本身并没有关系，而是操作系统和 CPU 协同工作的结果。正是因为操作系统和 CPU 努力地为每一个进程分配独立的地址空间，所以就算我毛手毛脚、糊里糊涂地制造了一个 bug，破坏了某个内存区域，顶多也就是让当前的应用程序趴窝，但不会影响其他进程*。

* Windows 95/98 时代，经常出现由于应用程序的异常导致操作系统瘫痪的情况……想不到操作系统也会有这样的硬伤，唉。

当然了，真正去保存内存数据的还是物理内存。操作系统负责将物理内存分配给虚拟地址空间，同时还会对每一个内存区域设定“只读”或者“可读写”等属性。

通常，因为程序的执行代码是只读的，所以有时候会和其他进程共享物理内存*。另外，当我们启动了几个笨重的应用程序而使内存出现不足时，操作系统把物理内存中还没有被引用的部分倒腾出来，保存到硬盘上。当程序再次需要引用这个区域的数据的时候，再从磁盘写回到内存（恐怕会把别的部分从磁盘里面取出来也说不定哦）。这个操作完全是在操作系统的后台进行的，对于应用程序来说，压根儿不知道背后发生的事。这时硬盘“咔嗒咔嗒”响，机器的反应也慢了下来。

* 现在，将程序的一部分以**共享库**的形式来共享使用，已经是很普遍的设计手法了。在写入之前，连内存数据也可以共享。

之所以能够这样，多亏了虚拟地址。正是因为避免了让应用程序直接面对物理内存的地址，操作系统才能够顺利地对内存区域进行重新配置（参照图 2-2）。

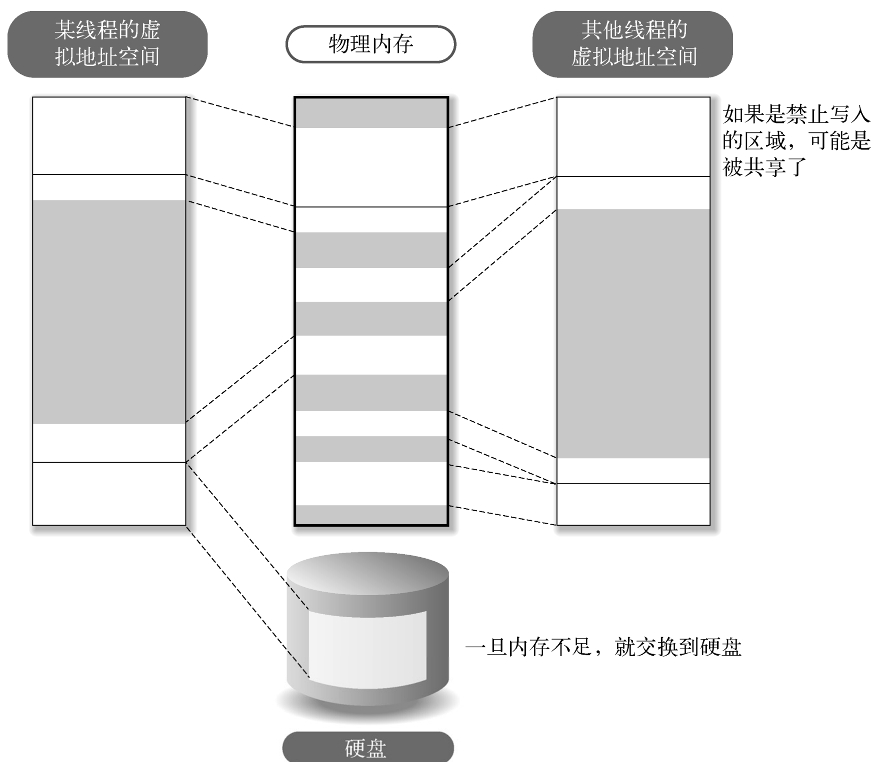


图 2-2　虚拟内存的概念图

> 要　点
> **在如今的运行环境中，应用程序面对的是虚拟地址空间。**



补充 关于 scanf()

代码清单 2-1 中，使用下面的两条语句让用户输入整数值：

```
fgets(buf, sizeof(buf), stdin);
sscanf(buf, "%d", &hoge);
```

在一般的 C 入门书籍中，却经常看到以下写法：

```
scanf("%d", &hoge);
```

在这一次的例子中如果使用这种写法，程序是不会如愿运行起来的。因为这种写法在一开始漏掉了 getchar(),没有使程序执行进入输入等待状态。

这个问题是由 scanf()的自身实现造成的。

scanf()不是以行单位对输入内容进行解释，而是对连续字符流进行解释（换行字符也视为一个字符）。

scanf()连续地从流读入字符，并且对和格式说明符（%d）相匹配的部分进行变换处理。

例如，当格式说明符为%d 的时候，输入

```
123↲
```

从流中取得 123 部分的内容，并对它进行处理。换行符依旧会残留在流中。因此，后续的 getchar()会吞食这个留下的换行符。

此外，当 scanf()变换失败的时候（比如，尽管你指定了%d，但是输入的却是英文字符），scanf()会将导致失败的部分遗留在流中。

在读入过程中有几个对象被成功地变换，则 scanf()的返回值就为几。如果做一下错误检查，可能有人会写出下面的代码：

```
while (scanf("%d", &hoge) != 1) {
    printf("输入错误，请再次输入！");
}
```

分析一下上面的代码，我们就会知道，**一旦用户错误输入过一次，这段程序就会进入无限循环**。原因就是：错误输入的那部分字符串，将会被下一个 scanf()读到。

像代码清单 2-1 那样，如果将 fgets()和 sscanf()组合使用，就可以避免这个问题。

当然，一旦对 `fgets()`函数的第2个参数赋予超过指定长度的字符串，也是会出问题的。不过，如果像代码清单2-1这样指定了256个字符，对于自己使用的程序来说应该是足够了。

顺便说一下，在 `scanf()`中通过指定复杂的格式说明符，同样可以避免问题的发生。但我还是感觉不如使用 `fgets()`这种方式来得便利。

此外，为了解决这个问题，有人会使用 `fflush(stdin);`，其实这是个**错误的处理方法**。

`fflush()`是对输出流使用的，它不能用于输入流。标准中并没有定义用于输入流的 `fflush()`的行为。

2.2 C的内存的使用方法

2.2.1 C的变量的种类

C语言的变量具有区间性的作用域*。

* 在标准中，“作用域”（scope）和“连接”（linkage）是分别定义的，用语句块包围的是作用域，static 和 extern 分别控制静态连接和外部连接。
对于全局变量，作用域指文件作用域，链接指外部链接。
对于程序员来说，这些方式都是控制命名空间的，它们没有什么不同。在本书中，我们统一使用“作用域”这种叫法。

在开发一些小程序的时候，也许我们并不在意作用域的必要性。可是，当你书写几万行，甚至几十万行的代码的时候，没有作用域肯定是不能忍受的。

C语言有如下三种作用域。

❶ 全局变量

在函数之外声明的变量，默认地会成为全局变量。全局变量在任何地方都是可见的。当程序被分割为多个源代码文件进行编译时，声明为全局变量的变量也是可以从其他源代码文件中引用的。

❷ 文件内部的静态变量

就算对于像全局变量那样被定义在函数外面的变量，一旦添加了 `static`，作用域就只限定在当前所在的源代码文件中。通过 `static` 指定的变量（包括函数），对于其他源代码文件是不可见的。在英语中，`static` 是“静态的”的意思，我实在想不明白为什么这个功能莫名其妙地被冠以“static”，这一点可以算是C语言的一个未解之谜。

❸ 局部变量

局部变量是指在函数中声明的变量。局部变量只能在包含它的声明的语句块（使用`{}`括起来的范围）中被引用。

局部变量通常在函数的开头部分进行声明，但也可以在函数内部某语句块的开头进行声明。例如，在“交换两个变量的内容时，需要使用一下临时变量”的情况下，将局部变量声明放在当前语句块开头还是比较方便的。

局部变量通常在它所在的语句块结束的时候被释放。如果你不想释放某个局部变量，可以在局部变量上加上 `static` 进行声明（在后面有详细说明）。

另外，除了作用域不同，C 的变量之间还有**存储期**（storage duration）的差别。

❶ 静态存储期（static storage duration）

全局变量、文件内的 `static` 变量、指定 `static` 的局部变量都持有**静态存储期**。这些变量被统称为**静态变量**。

持有静态存储期的变量的寿命从程序运行时开始，到程序关闭时结束。换句话说，静态变量一直存在于内存的同一个地址上。

❷ 自动存储期（auto storage duration）

没有指定 `static` 的局部变量，持有**自动存储期**。这样的变量被称为**自动变量**。

持有自动存储期的变量，在程序运行进入它所在的语句块时被分配以内存区域，该语句块执行结束后这片内存区域被释放*。

* 如果说明得细致一些，在几乎所有的处理环境中，并不是“程序执行进入语句块时”给自动变量分配内存区域，而是在“程序执行进入函数时”统一地进行内存区域分配的。

这个特征通常使用“栈”的机制来实现。2.5 节中会对此做详细说明。

接下来就不是“变量”了。C 中可以使用 `malloc()` 函数动态分配内存。通过 `malloc()` 动态分配的内存，寿命一直延续到使用 `free()` 释放它为止。

在程序中，如果需要保持一些数据，必须在内存中的某个场所取得相应大小的内存区域。总结一下，在 C 中有三种内存区域的寿命。

❶ 静态变量

寿命从程序运行时开始，到程序关闭时结束。

❷ 自动变量

寿命到声明该变量的语句块被执行结束为止。

❸ 通过 `malloc()` 分配的领域

寿命到调用 `free()` 为止。

要 点

C中有三种内存领域的寿命。

- ❑ 静态变量的寿命从程序运行时开始，到程序关闭时结束。
- ❑ 自动变量的寿命到声明该变量的语句块执行结束为止。
- ❑ 通过 malloc()分配的领域的寿命到调用 free()为止。



补充 **存储类型修饰符**

在C的语法中，以下关键字被定义为“存储类型修饰符”。

typedef extern static auto register

可是，在这些关键字中，真正是“指定存储区间”的关键字，只有static。*

extern 使得在其他地方定义的外部变量可以在本地可见；auto 是默认的，所以没有显式指定的必要；register 可以给出编译器优化提示（如今的编译已经很先进了，所以一般也不会使用这个关键字）；至于 typedef，它只是因为可以给编码带来便利才被归纳到存储类型修饰符中来的。

希望大家不要被这众多的“存储类型修饰符”搞得手忙脚乱。

* 可是，当你在函数的外面使用 static 的时候，就是使用作用域来控制了，而不是使用存储期。

2.2.2 输出地址

正如之前所说，C的变量中有几个阶段的作用域，而且变量之间还有“存储期”的区别。此外，通过 malloc()可以动态分配内存。

在内存中，这些变量究竟是怎样配置的呢？不如让我们来写个测试程序验证一下（参照代码清单 2-2）。

代码清单 2-2 print_address.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int              global_variable;
 5:  static int       file_static_variable;
 6:
 7:  void func1(void)
 8:  {
 9:      int func1_variable;
10:      static int func1_static_variable;
11:
```

```
12:     printf("&func1_variable..%p\n", &func1_variable);
13:     printf("&func1_static_variable..%p\n", &func1_static_
        variable);
14: }
15:
16: void func2(void)
17: {
18:     int func2_variable;
19:
20:     printf("&func2_variable..%p\n", &func2_variable);
21: }
22:
23: int main(void)
24: {
25:     int *p;
26:
27:     /*输出指向函数的指针*/
28:     printf("&func1..%p\n", func1);
29:     printf("&func2..%p\n", func2);
30:
31:     /*输出字符串常量的地址*/
32:     printf("string literal..%p\n", "abc");
33:
34:     /*输出全局变量*/
35:     printf("&global_variable..%p\n", &global_variable);
36:
37:     /*输出文件内的 static 变量的地址*/
38:     printf("&file_static_variable..%p\n", &file_static_
        variable);
39:
40:     /*输出局部变量*/
41:     func1();
42:     func2();
43:
44:     /*通过 malloc 申请的内存区域的地址*/
45:     p = malloc(sizeof(int));
46:     printf("malloc address..%p\n", p);
47:
48:     return 0;
49: }
```

在我的环境中运行结果如下：

```
&func1..0x8048414
&func2..0x8048440
string literal..0x8048551
&global_variable..0x804965c
&file_static_variable..0x8049654
&func1_variable..0xbfbfd9d8
&func1_static_variable..0x8049650
&func2_variable..0xbfbfd9d8
malloc address..0x805b030
```

一开始我们说要将变量的地址输出，代码清单 2-2 的第 28 ~ 29 行却输出了**指向函数的指针**。

尽管到目前为止没有提到过函数指针的问题，但是这次的代码中出现了函数指针。函数通过编译器解释成机器码，并且被配置在内存的某个地方的地址上。

在 C 中，正如数组在表达式中可以被解读成指针一样，“函数”也同时意味着“指向函数的指针”。通常，这个指针指向函数的初始地址。

第 32 行输出使用""包围的字符串（字符串常量）的地址。

在 C 中，“字符串”是作为“char 的数组”来表现的。字符串常量类型也是“char 的数组”，因为**表达式中的数组可以解读成“指向初始元素的指针”**，所以表达式中的“abc”，同样也意味着保存这个字符串内存区域的初始地址。

字符串常量在 C 中也被做了特别对待，它总让人感觉“不知道它被保存在内存的哪一片区域”，所以在这里我们也尝试输出它的地址。

第 35 行和第 38 行，分别输出了全局变量的地址和文件内 static 变量的地址。

第 41 行和第 42 行，调用了函数 func1()和 func2()。第 12 行和第 20 行输出自动变量的地址，第 13 行输出 static 局部变量的地址。

再回到 main()函数，在第 46 行输出利用 malloc()分配的内存区域的地址。

接下来，让我们来观察一下实际被输出的地址。

乍一看，都是 0x80……、0xbf……这样的地址，有点晕。

将地址按照顺序重新排列，并且整理成表 2-1。

表2-1　地址一览表

地　址	内　容	地　址	内　容
0x8048414	函数func1()的地址	0x804965c	全局变量
0x8048440	函数func2()的地址	0x805b030	利用malloc()分配的内存区域
0x8048551	字符串常量	0xbfbfd9d8	func1()中的自动变量
0x8049650	函数内的static变量	0xbfbfd9d8	func2()中的自动变量
0x8049654	文件内static变量		

通过观察，我们发现“指向函数的指针”和“字符串常量”被配置在非常近的内存区域。此外，函数内 static 变量、文件内 static 变量、全局变量等这些静态变量，也是被配置在非常近的内存区域。接下来就是 malloc()分配的内存区域，它看上去和自动变量的区域离得很远。最后你可以发现，func1()和 func2()的自动变量被分配了完全相同的内存地址。

如果使用图来说明，应该是下面这样的感觉（参照图 2-3）。

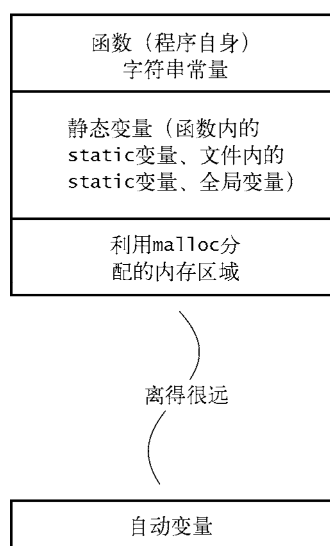


图 2-3　各种各样的地址

在后面的章节中，我们将会逐一对这些内存区域进行详细的说明。

2.3　函数和字符串常量

2.3.1　只读内存区域

在我的处理环境中，函数（程序自身）和字符串常量被配置在内存里相邻的地址上。

这并不是偶然的，如今的大多数操作系统都是将函数自身和字符串常量汇总配置在一个**只读内存区域**的。

由于函数本身不可能需要改写，所以它被配置在内存的只读区域。其实在很久以前，机器语言的程序改写自身程序代码的技术是被大量使用的*，但是现在的操作系统几乎都禁用了这种技术。

* 笔者就曾经有这样的经历。在 Z80 的情况下，只能通过直接指定地址的方式调用子程序……当然这些都是老皇历了。

那些能够修改自身代码的程序代码是非常晦涩难懂的。此外，如果执行程序是只读的，在同一份程序被同时启动多次的时候，通过在物理地址上共享程序能够节约物理内存。此外，由于硬盘上已经存放了可执行程序，就算内存不足，也不需要将程序交换到虚拟内存，相反可以将程序直接从内存中销毁*。

* 大体上，Windows、UNIX 等操作系统就是这样实现的。因此，在一部分 UNIX 中，如果在程序运行的时候将该程序改写（再编译/连接等），当前运行的程序就会崩溃。在如今的处理环境中，经常会对程序加锁。

根据处理环境的不同，字符串常量也有可能被配置在可改写的内存区域。像 DOS 这样不实施内存保护的操作系统也就罢了，可是在 UNIX 中，也有将字符串常量配置在非只读内存区域的情况。

假设有下面这样一个函数：

```
void func(void)
{
    char *str = "abc";

    printf("str..%s\n", str);
          ⋮  省略的很多逻辑
    str[0] = 'd';
}
```

一旦允许改写字符串常量，第一次调用函数输出“abc”，第二次调用函数却会输出“dbc”。可是根据代码的逻辑，给 str 赋值“abc”后，紧接着就会输出 str 的值。尽管如此，第二次调用还是输出了“dbc”，这就很让人头大。

2.3.2　指向函数的指针

函数可以在表达式中被解读成“指向函数的指针”，因此，正如代码清单 2-2 的实验那样，写成 func 就可以取得指向函数的指针。

“指向函数的指针”本质上也是指针（地址），所以可以将它赋给指针型变量。

比如有下面的函数原型：

```
int func(double d);
```

保存指向此函数的指针的变量的声明如下：

```
int (*func_p)(double);
```

然后写成下面这样，就可以通过 func_p 调用 func，

```
int (*func_p)(double);     ←声明

func_p = func;     ←将 func 赋给 func_p

func_p(0.5);     ←此时，func_p 等同于 func
```

将“指向函数的指针”保存在变量中的技术经常被运用在如下场合：

- GUI 中的按钮控件记忆“当自身被按下的时候需要调用的函数”
- 根据“指向函数的指针的数组”对处理进行分配

后者的“指向函数的指针的数组”，像下面这样使用：

```
int (*func_table[])(double) = {
    func0,
    func1,
    func2,
    func3,
};
    :
    :
    :
func_table[i](0.5);     ←调用 func_table[i]的函数，参数为 0.5
```

使用上面的写法，不用写很长的 switch case，只需通过 i 的值就可以对处理进行分配。

哦？不明白为什么？

确实，像

```
int (*func_p)(double);     ←指向函数的指针
```

还有，

```
int (*func_table[])(double);     ←指向函数的指针的数组
```

这样的声明，是不能用普通的方法来读的。

关于这种声明的解读方式，会在第 3 章进行说明。

2.4 静态变量

2.4.1 什么是静态变量

静态变量是从程序启动到运行结束为止持续存在的变量。因此，静态变量总是在虚拟地址空间上占有固定的区域。

静态变量中有全局变量、文件内 static 变量和指定 static 的局部变量。因为这些变量的有效作用域各不相同，所以编译和连接时具有不同的意义，但是**运行的时候它们都是以相似的方式被使用的**。

2.4.2 分割编译和连接

在 C 语言中，一个程序可以由多个源代码文件构成，并且这些源代码文件在各自编译之后可以连接起来。这一点对于大规模的编程工作来说，是举足轻重的。难道不是吗？如果 100 个程序员一拥而上，一起捣鼓同一个源代码文件，那真是不可想象。

此外，关于函数和全局变量，如果**它们的名称相同**，即使它们跨了多个源代码文件也被作为相同的对象来对待。进行这项工作的是一个被称为“链接器”的程序。

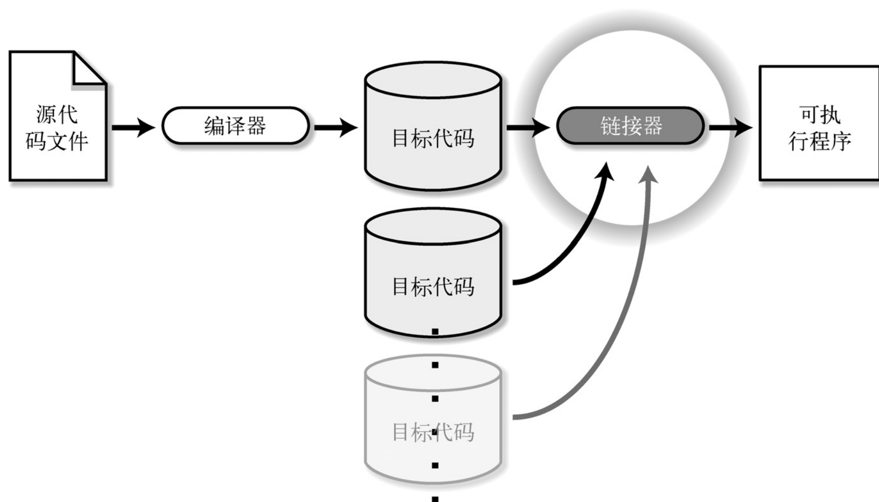


图 2-4 链接器

为了在链接器中将名称结合起来，各目标代码大多都具备一个**符号表**（symbol table）（详细内容需要依赖实现细节）。比如在 UNIX 中，可以使用 nm 这样的命令窥视符号表的内容。

不好意思，这里是 UNIX 特有的话题。为了测试，我们使用 cc -c 编译 print_address.c（参照代码清单 2-2），生成 print_addres.o 文件，之后对这个文件使用 nm 命令。在我的环境中，结果输出如下。

```
> cc -c print_address.c
> nm print_address.o
00000004 b file_static_variable
00000000 T func1
00000000 b func1_static_variable.4
0000002c T func2
00000000 t gcc2_compiled.
00000004 C global_variable
00000048 T main
         U malloc
         U printf
```

通过观察输出结果，我们很容易发现符号表中记录了文件内的 static 变量、局部 static 变量这些看上去不需要连接的对象。

如果命名空间不一样，确实不需要和其他文件的符号连接，但是对于静态变量来说，因为必须要给它们分配一些地址，所以符号表中记录了这些变量。可是同时我们也发现全局变量的标记有些特别。与全局变量使用 C 进行标记不同的是：和外部没有连接的符号，无论是局部的还是文件内部的 static 变量，都使用了 b 进行标记。

局部 static 变量 func1_static_variable 的后面被追加了 .4 这样的标记，这是因为在同一个 .o 文件中，局部 static 变量的名称有可能会发生重复，所以在它后面追加了识别标记。

函数名后面追加了 T 或者 U。如果函数是在当前文件中定义的，就在其函数名后加 T；如果函数定义在当前文件之外，只是在当前文件内部调用此函数，就在此函数后面加 U。

gcc2_compiled. 是我们没有见过的符号，你可以把它当成处理环境随意加上的标记，把它放在一边。

链接器就是根据这些信息，给这些到目前为止还只是个“名称”的对象分配地址*。

* 现在对共享库做动态链接变成一件很自然的事，而现实中可没有这么单纯……

请注意自动变量完全没有出现在符号表中。这是因为自动变量的地址是在运行时被决定的，它属于链接器管辖范围以外的对象。关于这一点，我们在下一节阐述。

2.5　自动变量（栈）

2.5.1　内存区域的“重复使用”

通过代码清单 2-2 的实验，我们看到 func1()的自动变量 func1_variable 和 func2()的自动变量 func2_variable 存在于完全相同的地址上。

在声明自动变量的函数执行结束后，自动变量就不能被使用了。因此，func1()执行结束后，func2()重复使用相同的内存区域是完全没有问题的。

> 要　点
> **自动变量重复使用内存区域。**
> **因此，自动变量的地址是不一定的。**

2.5.2　函数调用究竟发生了什么

自动变量在内存中究竟是怎样被保存的？为了更加详细地了解这一点，还是让我们用下面这个测试程序做一下实验。

代码清单 2-3　auto.c

```
 1:  #include <stdio.h>
 2:
 3:  void func(int a, int b)
 4:  {
 5:      int c, d;
 6:
 7:      printf("func:&a..%p &b..%p\n", &a, &b);
 8:      printf("func:&c..%p &d..%p\n", &c, &d);
 9:  }
10:
11:  int main(void)
12:  {
13:      int a, b;
14:
15:      printf("main:&a..%p &b..%p\n", &a, &b);
16:      func(1, 2);
17:
18:      return 0;
19:  }
```

在我的环境中运行结果如下：

```
main:&a..0xbfbfd9e4 &b..0xbfbfd9e0
func:&a..0xbfbfd9d8 &b..0xbfbfd9dc
func:&c..0xbfbfd9cc &d..0xbfbfd9c8
```

如果把运行结果用图来说明，应该是图 2-5 这样的。

如果将 main()的局部变量、func()的局部变量以及 func()的形参的地址进行比较，func()相关的地址看上去相对小一些。

C 语言中，在现有被分配的内存区域之上以“堆积”的方式，为新的函数调用分配内存区域*。在函数返回的时候，会释放这部分内存区域供下一次函数调用使用。图 2-6 粗略地表现了这个过程。

* 图 2-3 中，自动变量的区域上方有一片广大的区域。在这片区域中，栈不断地增长。

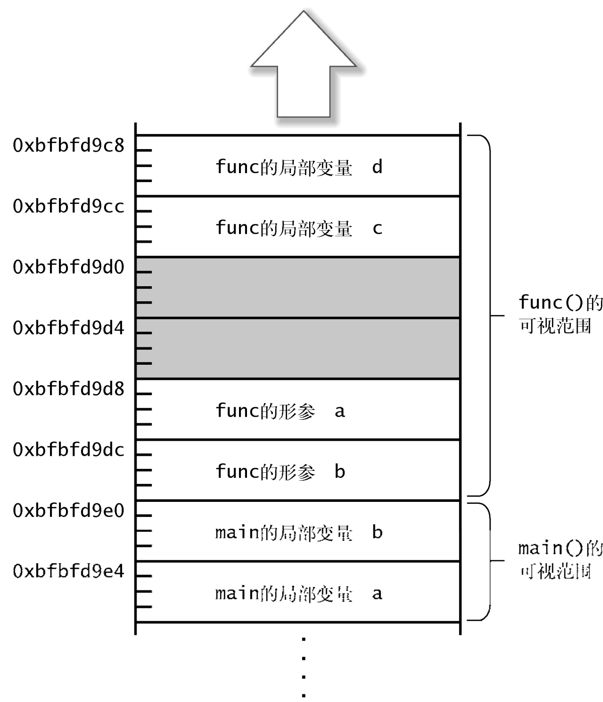


图 2-5　局部变量和参数的地址

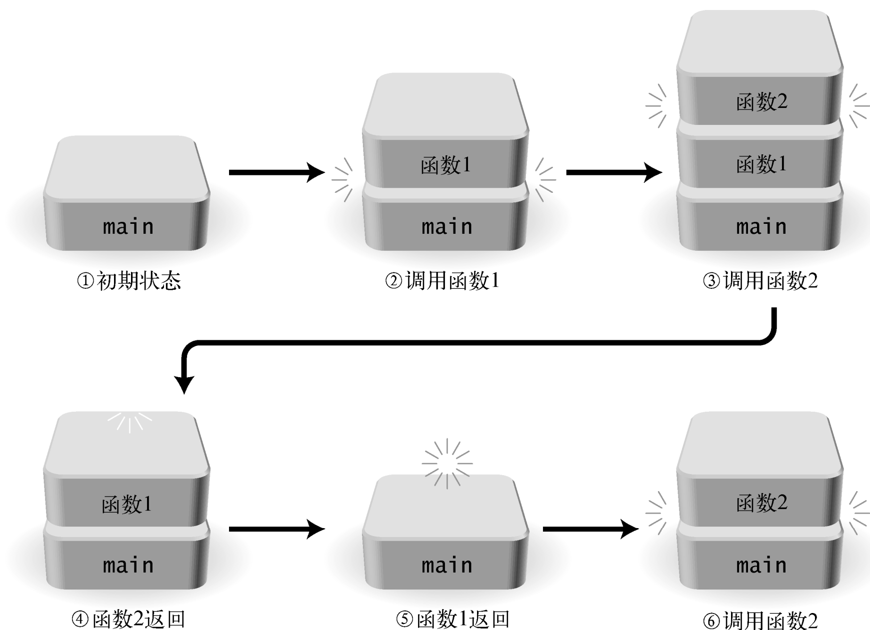


图 2-6　函数调用的概念图

对于像这样使用“堆积”方式的数据结构，我们一般称为**栈**。

程序员们有时候也使用数组等方式实现栈。但是，大部分的 CPU 中已经嵌入了栈的功能，C 语言通常直接使用。

> 要　点
> **C 语言中，通常将自动变量保存在栈中。**

通过将自动变量分配在栈中，内存区域可以被重复利用，这样可以节约内存。

此外，将自动变量分配在栈中，对于递归调用（参照 2.5.4 节）也具有重要的意义。

下面归纳了最简约的 C 语言函数调用的实现*：

❶ 在调用方，参数**从后往前**按顺序被堆积在栈中*。

❷ 和函数调用关联的返回信息（返回地址等）也被堆积在栈中（对应于图 2-5 的灰色部分）。所谓的“返回地址”，是指函数处理完毕后应该返回的地址。正因为返回地址被堆积在栈中，所以无论函数从什么地方被调用，它都能返回到调用点的下一个处理。

* 当然，最近的编译器做了各种优化，它们不一定完全以这里描述的方式动作。但是基本思想是一致的。

* 关于为什么参数是从后往前堆积的，请参照 2.5.3 节。

❸ 跳转到作为被调用对象的函数地址。

❹ 栈为当前函数所使用的自动变量增长所需大小的内存区域。❶到❹所增长的栈的区域成为当前函数的可引用区域。

❺ 在函数的执行过程中，为了进行复杂的表达式运算，有时候会将计算过程中的值放在栈中。

❻ 一旦函数调用结束，局部变量占用的内存区域就被释放，并且使用返回信息返回到原来的地址。

❼ 从栈中除去调用方的参数。

图 2-7 展示了 `func(1, 2)`被调用时，栈的使用情况。它与图 2-5 相吻合。

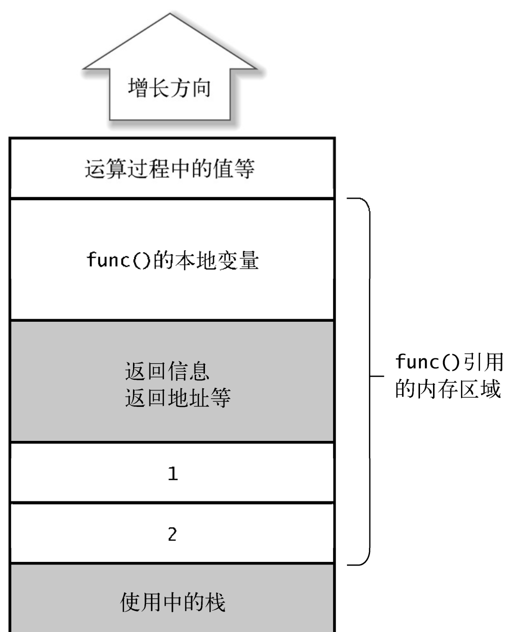
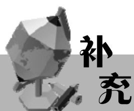


图 2-7　调用 `func(1, 2)`时栈的使用情况

此外，在调用函数时，请留意为形参分配新的内存区域。我们经常听说“C 的参数都是传值，向函数内部传递的是实参的副本”其中的复制动作，其实就是在这里发生的。



补充 一旦函数执行结束，自动变量的内存区域就会被释放！

初学者经常写出下面这样的程序。

```
/*将int 变换成字符串的程序*/
char *int_to_str(int int_value)
{
    char  buf[20];

    sprintf(buf, "%d", int_value);

    return buf;
}
```

恐怕……这个程序不能正常地跑起来*。

原因估计你也知道了：**自动变量 buf 的内存区域在函数执行结束后就会被释放。**

为了让这个程序先动起来，可以将 buf 声明成下面这样：

```
static char buf[20];
```

因为 buf 的内存区域一直会被静态地保持，所以即使函数执行结束，buf 的内存区域也不会被释放。

在这种方式下，当你连续两次调用这个函数，第二次函数调用会“悄悄地”修改第一次函数调用得到的字符串。

```
str1 = int_to_str(5);
str2 = int_to_str(10);
printf("str1..%s, str2..%s\n", str1, str2);  ←这里究竟会输出什么样的结果呢?
```

程序员当然希望输出的是 str1..5，str2..10。可惜最终事与愿违。

这样的函数引起了一个出乎意料的 bug。此外，在多线程编程的情况下，当前函数同样也是有问题的*。

借助于 malloc()动态地分配内存区域，可以解决上面的问题（参照 2.6 节），但是使用方需要同时使用 free()*。

其实，作为最终的解决方案，如果函数调用方事先知道数组的长度上限，可以在调用方声明一个数组，然后通过函数将结果输出到此数组中。对于 C 语言来说，这是一个最实用的方案。

* 在某些环境下可能也能跑起来，但这只是偶然。

* 标准库中有一个函数 strtok()，此函数也存在类似的问题，招致了很多抱怨。

* 由于 Java 具有垃圾回收机制，所以不需要我们显式地调用 free()。



补充　一旦破坏了自动变量的内存区域

假设，通过自动变量声明下面的数组：

```
int  hoge[10];
```

如果没有做数组长度检查，将数据写入了超过数组内存区域的地方，究竟又会引发怎样的悲剧呢？

如果只是超过一点点，可能也就是破坏相邻的自动变量的内容。但是，一旦将前方的内存区域径直地破坏下去……

自动变量是保存在栈中的，如果是数组，表示如下图（参照图 2-8），

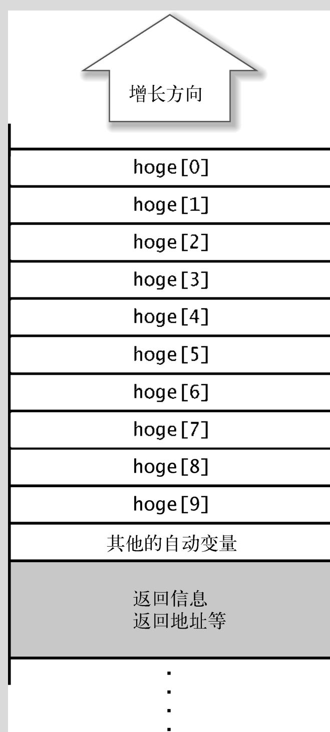


图 2-8　栈中的数组

在这里，如果大范围地超过数组内存区域写入数据，内存中直到存储函数的返回信息的区域都有可能被破坏。也就是说，这个函数**不能返回了**。

当我们在追踪一个 bug 的时候，发现函数处理已经走到了最后，可偏偏不能返回到调用方。此时，就应该质疑是不是已经不小心将数据写入了保存函数返回地址的内存领域。

这种情况下，就算是调试也经常不能定位程序发生错误的地方。因为调试也是使用堆积在栈中的信息，如果大量地破坏了栈的内存领域，是很难追踪到 bug 发生的具体地点的。

此外，由于越界向数组类型的自动变量写入数据，以至于返回信息（返回地址）被覆盖，甚至可能造成安全漏洞。

对于那些没有切实做好数组长度检查的程序，一些无良的骇客会故意地传递很大的数据，造成函数的返回地址被恶意的数据所改写*。也就是说，骇客完全可以让你的程序去运行任意的机器语言代码。

* 请参照"'黑客不是骇客'协会"http://www-vacia.media.is.tohoku.zc.jp/~s-yamane/hackerML/。

标准库中有一个 gets()函数，它和 fgets()同样都可以从标准输入读取一行输入，但与 fgets()不同的是你不能向它传递缓冲的大小。因此，gets()是**不可能做数组长度检查**的。在 C 语言中，所谓将数组作为参数进行传递，只不过是传递指向数组初始元素的指针。作为被调用方，是完全不知道数组究竟有多长的。

gets()函数多用于将标准输入，也就是"从外部"来的输入保存在数组中。因此，在使用 gets()的程序中，通过故意将包含尺寸很大的行的数据传递给 gets()，就可以达到数组越界且改写返回地址的目的。

1988 年名震互联网的"互联网蠕虫"，就是利用了 gets()的这个弱点。

因为这些原因，gets()已经被视为落后于时代的函数。我使用的编译器（gcc version 2.7.2.1）在编译和连接过程中，一旦发现当前代码使用了 gets()，就会提示以下警告：

```
warning: this program uses gets(), which is unsafe.
```

如果你一意孤行还是要运行这个程序，还会提示同样的警告。

不只是 gets()，还有比如对于 scanf()这个函数，如果使用"%s"也会招致同样的结果。但是，你可以通过对 scanf()指定"%10s"来限制字符串的最大长度。

下面再给大家举一个有点恶搞的例子（参照代码清单 2-4）。

并不是所有的运行环境都是这样，但是至少在 Windows 98 和 FreeBSD（编译器是 gcc）下，hello()会被调用（而且不止一次）。

为什么会这样？——请读者自己思考。

另外，这个程序必然会以崩溃告终，所以请在具有内存保护的操作系统上运行。

代码清单 2-4　stackoverflow.c

```
 1:  #include <stdio.h>
 2:
 3:  void hello(void)
 4:  {
 5:      fprintf(stderr, "hello!\n");
 6:  }
 7:
 8:  void func(void)
 9:  {
10:      void        *buf[10];
11:      static int  i;
12:
13:      for (i = 0; i < 100; i++) {     ←—越界!
14:          buf[i] = hello;
15:      }
16:  }
17:
18:  int main(void)
19:  {
20:      int buf[1000];
21:
22:      func();
23:
24:      return 0;
25:  }
```

2.5.3　可变长参数

大部分的C语言入门书籍往往在一开始就频繁地使用`printf()`这个输出文字信息的函数，利用这个函数，可以将可变个数的参数填充到字符串中的指定位置。因为这个奇怪的特征，在教初学者 C 语言的时候，经常会有这样让人费解的说明：

> 快看，在 C 中的输入输出并不是语言的一部分，而是作为函数库被提供的。因此，`printf()`这个函数和程序员平时写的函数没有什么不一样哦。

在这里姑且把这个问题先放一边。

正如 2.5.2 节中说明的那样，C 语言的参数是从后往前被堆积在栈中的。

另外，在 C 语言中，应该是由调用方将参数从栈中除去。

顺便提一下，Pascal 和 Java 是从前往后将参数堆积在栈中的。这种方式能够从前面开始对参数进行处理，所以对于程序员来说比较直观。此外，将参数从栈中除去是被调用方应该承担的工作。大部分情况下，这种方式的效率还是比较高的。

为什么 C 故意采取和 Pascal、Java 相反的处理方式呢？其实就是为了实现可变长参数这个功能*。①

* Pascal 和 Java 中不能写带有可变长参数的函数。

比如，对于像

```
printf("%d, %s\n", 100, str);
```

这样的调用，栈的状态如图 2-9 所示。

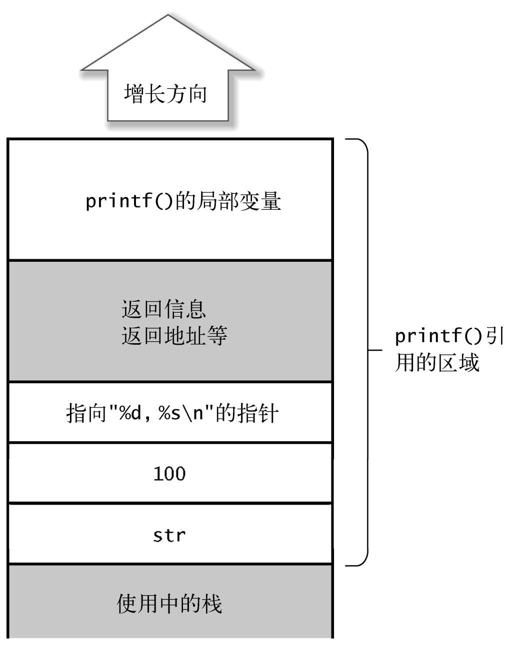


图 2-9　调用持有可变长参数的函数

重要的是，无论需要堆积多少个参数，**总能找到第一个参数的地址**。从图中我们可以看出，从 `printf()` 的局部变量来看，第一个参数（指向 `"%d, %s\n"` 的指针）一定**存在于距离固定的场所**。如果从前往后堆积参数，就肯定不能找到第一个参数。

① Java 在 JDK1.5 之后的版本也开始支持可变长函数了。——译者注

之后，在 printf()中，通过对找到的第一个参数"%d, %s\n"进行解析，就可以知道后面还有几个参数。

由于其余的参数是连续排列在第一个参数后面的，所以你可以顺序地将它们取出。

可是，现实中有一些部分是依赖运行环境的。ANSI C 为了提高可移植性，通过头文件 stdarg.h*提供了一组方便使用可变长参数的宏。

* ANSI C 以前的 C 使用 varargs.h 这个头文件，它和 stdarg.h 在使用上有很大的差异。

下面就让我们实际使用 stdarg.h 写一个具有可变长参数的函数。

我们考虑写一个山寨版的 printf()，取名为 tiny_printf()。

tiny_printf()的第一个参数指定后续的各参数的类型，第二个参数开始指定需要输出的值。

```
tiny_printf("sdd", "result..", 3, 5);
```

在这个例子中，通过第一个参数"sdd"，指定后续的参数类型为“字符串，int，int”（和 printf()一样，s 表示字符串，d 表示数字）。从第二个参数开始，分别向函数传递字符串"result"和两个整数。

函数执行后的结果如下：

```
result.. 3 5
```

和 printf()不同，因为指定换行字符的输出比较繁琐，tiny_printf()在默认情况下会主动进行换行。

代码如下（参照代码清单 2-5）。

代码清单 2-5　tiny_printf.c

```
 1:  #include <stdio.h>
 2:  #include <stdarg.h>
 3:  #include <assert.h>
 4:
 5:  void tiny_printf(char *format, ...)
 6:  {
 7:      int i;
 8:      va_list     ap;
 9:
10:      va_start(ap, format);
11:      for (i = 0; format[i] != '\0'; i++) {
12:          switch (format[i]) {
13:            case 's':
```

```
14:                printf("%s ", va_arg(ap, char*));
15:                break;
16:              case 'd':
17:                printf("%d ", va_arg(ap, int));
18:                break;
19:              default:
20:                assert(0);
21:            }
22:        }
23:        va_end(ap);
24:        putchar('\n');
25:    }
26:
27:    int main(void)
28:    {
29:        tiny_printf("sdd", "result..", 3, 5);
30:
31:        return 0;
32:    }
```

从第 5 行开始函数的定义。对于形参声明中的…这种写法，可能大家会感到有些陌生，但原型声明也是写成这样的。如果原型声明的参数中出现…，对于这部分的参数是不会做类型检查的。

在第 8 行，声明 va_list 类型的变量 ap。stdarg.h 文件中定义了 va_list 类型，我的环境中是这样定义的，

```
typedef char *va_list;
```

也就是说，在我的环境中，va_list 被定义成“指向 char 的指针”。

在第 10 行，va_start(ap, format);意味着“使指针 ap 指向参数 format 的下一个位置”。

因此，我们得到了可变长部分的第一个参数。在后面的第 14 行和第 17 行，给宏 va_arg()指定 ap 和参数类型，就可以顺序地取出可变长部分的参数。

第 23 行的 va_end()只不过是个“摆设”，只不过因为在标准里指出了对于具有 va_start()的函数需要写 va_end()。

顺便说一下，在我的环境中，宏 va_end()是这样被定义的：

```
#define        va_end(ap)
```

实际上，va_end()就是个空定义。

以上就是开发具有可变长参数的函数的方法。

这里必须要注意的是，因为无论怎样都是从函数内部顺序地读出各参数，所以要实现可变长参数的函数就必须知道参数的类型和个数。

printf()将输出内容整理得很利落，但是如果只是输出很少的内容，使用printf()就显得有点小题大做。这种情况下，Pascal的writeln()和BASIC的print语句倒是比较简洁。

如

```
writeln("a..", 10, " b..", 5);
```

输出结果为：

```
a..10 b..5
```

但是C语言是**无法提供这样的函数的**。这是因为在writeln()内部无法知道参数的类型和个数。

话说回来，一旦学会了可变长函数的开发方法，还是感觉自己挺了不起的，无论如何总是想找个机会秀一下。我就是这样的*。

可是，对于可变长参数的函数，是不能通过原型声明来校验参数的类型的。另外，函数的执行需要被调用方完全信任调用方传递的参数的正确性*。因此，对于使用了可变长参数的函数，调试会经常变得比较麻烦。一般只有在这种情况下，才推荐使用可变长参数的函数：如果不使用可变长参数得函数，程序写起来就会变得困难。

* 以前年少无知，看到XView的函数使用方法，还觉得挺帅的。

* 偶尔，对于printf，编译器有时也会亲切地给出“格式定义符和实际参数的类型不一致”的警告。其实这只不过是编译器特别照顾了printf()。毕竟printf()是一个被“超”频繁使用的函数。

> 要　点
> **在决定开发可变长参数的函数之前，有必要讨论是否真的需要这么做。**



补充　assert()

代码清单2-5中，有assert(0);这样的语句。

assert()是在assert.h中定义的宏，使用方式如下：

```
assert(条件表达式);
```

若条件表达式的结果为真，什么也不会发生；若为假，则会输出相关信息并且**强制终止程序**。

我们经常会看到如下注释：

```
/*这里的str[i]必定为'\0' */
```

虽然这种方式可以提高程序的可读性，但也不是说你可以什么检查也不做。这种情况下，如果使用，

```
assert(str[i] == '\0');
```

可以帮我们挑出bug。

代码清单2-5的assert(0)的参数为0（假），因此只要程序执行经过这里就会被强制终止。如果程序自身没有bug，程序是不会走到过程语句switch的default部分的，我认为这种编程方式是值得肯定的。

对于使用“强制终止程序运行”的方式来进行异常处理，很多人是持有反对意见的。确实，对于用户的一些奇怪的操作，比如传入一个奇怪的文件等行为，突然草率地终止程序会让用户感到不知所措。

可是，如果不是由于这样的“外部因素”，只要**程序自身没有bug就绝对不会发生异常**的情况下，我想还是应该果断地终止程序。使用“通过返回值返回错误状态”等冗长的处理方式，一旦调用方偷懒没做检查*，就会放过很多本来可以很容易就发现的bug。

像C这样可以导致内存区域破坏的语言，一旦发现很明显的bug，就表明当前程序已经无法保证正常运行了。如果栈被破坏了，就算你想返回错误状态，也有可能无济于事，因为当前的函数可能连return也完成不了*。

让我们在那些潜在bug还没有给你带来更大的麻烦之前，麻利地把它们消灭吧!*

* 这个经常有。☹

* 如果是Java这样能够保证内存区域不会破坏，并且具备完善的异常处理机制的语言，将例外返回反而是一个正确的做法。

* 如果是在编辑重要的数据，应该采取紧急安全措施。当然了，文件名称需要和普通情况下的不一样。



补充　尝试开发一个用于调试的函数吧

经常看到很多人为了调试方便，使用printf()输出变量的值*。

但如果直接使用printf()或者fprintf()，当调试结束后要删除它们可就麻烦了。

```
#ifdef DEBUG
    printf(...);
#endif /* DEBUG */
```

在有的书籍中推荐了上面的写法，但如果代码中充斥了大量这样的东西，就会变得很难读懂。

我们可能更需要下面这样类似于printf()的写法：

* “拜托你使用调试好吗？”我们经常会听到这样的声音。实际上由于这样那样的原因，我们不能完全否定“printf()调试”这样的做法。

```
debug_write("hoge..%d, piyo..%d\n", hoge, piyo);
```

可是，因为 printf()是持有可变长参数的函数，无法实现通过 debug_write()函数重新调用 printf()的功能。所以说，自己实现 printf()还是有些难度的。啊，怎么办呢？

这种情况下，标准库中提供了 vprintf()和 vfprintf()这两个函数。

```
void debug_write(char *fmt, ...)
{
    va_list   ap;
    va_start(ap, fmt);

    vfprintf(stderr, fmt, ap);    ←— 参数中传递 ap 的地方是指针

    va_end(ap);
}
```

此外，如果在此函数中引用一个全局变量，就可以用来控制是否输出调试信息。但伴随着 debug_write()调用而导致的开销是无法避免的。

如果是宏，在编译时是可以完全回避这部分系统开销的。但可惜的是，我们无法向 C 的宏传递可变长参数。

在下面，我们先定义一个宏：

```
#ifdef DEBUG
#define DEBUG_WRITE(arg)  debug_write arg
#else
#define DEBUG_WRITE(arg)
#endif
```

然后使用如下技巧：

```
DEBUG_WRITE(("hoge..%d\n", hoge));
```

这里必须要加两重括号，这可是个难点哟。

C 语言的常见问题集《C 语言编程 FAQ》[6]中介绍了，在非调试模式时将 DEBUG_WRITE 定义为(void)的技巧（p.151）*。

通过这个技巧，上面的语句会展开成(void)("hoge..%d\n", hoge)的形式，它意味着“将逗号运算符连接的表达式强制转型成 void”。优秀的编译器会通过优化完全忽略这条语句。

随便提一下，在 ISO C99 中的宏已经可以取得可变长参数。不过好像几乎都不是专门用于调试的。

或者，对于比较简单的输出，如果事先定义这样一个针对 int、double、char*的宏，可以使后面的开发变得简便一些。

```
#define SNAP_INT(arg) fprintf(stderr, #arg "...%d\n", arg)
```

* 非常遗憾，《C 语言编程 FAQ》的日语版（英文版名为 Cprogramming FAQS），已经绝版了。（第 8 次印刷注：由新纪元出版社重新出版。ISBN：4775302507）

这个宏可以像下面这样使用（如果有兴趣知道原理，你可以调查一下预处理器手册）：

```
SNAP_INT(hoge);
```

输出结果为：

```
hoge...5
```

此外还有一点需要补充。使用printf()输出调试信息，由于输出被缓冲，所以在程序异常结束的时候会发生关键信息没有被输出的情况。通过fprintf()将调试信息输出到stderr或者文件时*，建议事先使用setbuf()停止缓冲。

* 依照标准，stderr是不进行缓冲的。

2.5.4 递归调用

C语言通常在栈中分配自动变量的内存区域。这除了可以重复使用该区域来节省内存以外，还可以实现递归调用。

递归调用就是函数对自身的调用。

可是很多程序员对递归都感到比较纠结。

纠结的原因，除了递归调用本身的难度之外，也有“不明白究竟有什么作用”这一点。

世间的C语言入门书籍也经常用阶乘运算等作为递归调用的例题，我觉得这是有点不太合适的。为什么呢？阶乘什么的，**用循环来写不是更简单易懂吗**？

关于现实中递归调用的使用，就拿我来说，遍历树结构、图形结构中各元素的情况是占绝大多数的，但由于篇幅的限制，就不用这些应用作为例子了。

这里举一个“如果不用递归调用，程序就不太好写”的例子：快速排序。

所谓排序，就是将很多数据以一定的顺序（比如从小到大等）进行排列。作为排序的手法，冒泡算法、插入算法、堆排序等很多方法都为大家所知。顾名思义，快速排序是其中最快速的排序算法*。

* 实际上，利用像本书中举的这么简单的程序来应对特殊的案例，似乎显得过于幼稚。尽管如此，几乎在所有的情况下，处理速度还是比冒泡算法等要快很多。

快速排序的基本思路如下：

❶ 从需要排序的数据中，找到一个**适当的**基准值（pivot）。

❷ 将需要排序的数据按照小于 pivot 和大于 pivot 进行分类。
❸ 对分类后的两类数据各自再次进行上述的❶和❷的处理。

如果排列对象是数组，上面的第 2 个步骤就点麻烦。下面是在不使用大的临时内存区域的情况下，对数组进行分类的思路（假设是升序排序）：

❶ 从左向右，检索比 pivot 大的数据。
❷ 从右向左，检索比 pivot 小的数据。
❸ 如果两个方向都能检索到数据，将找到的数据交换。
❹ 重复进行❶～❸的操作，直到从左开始检索的下标和从右开始检索的下标发生冲突为止。

可能有人会对这种算法的敏捷性表示怀疑，但是这种算法的确很高效。在我的运行环境中，测试对 5 万个随机整数进行排序，冒泡算法排序花了 117 秒，快速排序算法用了 65 毫秒就完成了工作（1 毫秒=千分之一秒）。

> **要　点**
> **根据算法不同，程序的处理速度会有天壤之别。所以，选择合适的算法是非常重要的。**

代码清单 2-6 是实现快速排序的源代码。

代码清单 2-6　快速排序的程序

```
 1:  #define SWAP(a, b) {int temp; temp = a; a = b; b = temp;}
 2:
 3:  void quick_sort_sub(int *data, int left, int right)
 4:  {
 5:      int left_index = left;
 6:      int right_index = right;
 7:      int pivot = data[(left + right) / 2];
 8:
 9:      while (left_index <= right_index) {
10:          for ( ; data[left_index] < pivot; left_index++)
11:              ;
12:          for ( ; data[right_index] > pivot; right_index--)
13:              ;
14:
15:          if (left_index <= right_index) {
16:              SWAP(data[left_index], data[right_index]);
17:              left_index++;
18:              right_index--;
19:          }
20:      }
21:
22:      if (right_index > left) {
```

```
23:             quick_sort_sub(data, left, right_index);
24:         }
25:         if (left_index < right) {
26:             quick_sort_sub(data, left_index, right);
27:         }
28:     }
29:
30:     void quick_sort(int *data, int data_size)
31:     {
32:         quick_sort_sub(data, 0, data_size - 1);
33:     }
```

这个例程是对 int 的数组进行排序。

向函数 quick_sort()传递 int 型数组 data 和数组的长度，数组中的元素会被升序排序。

quick_sort_sub()将数组 data 的元素从 data[left]到 data[right]的部分进行排序（包括 data[left]和 data[right]）。

quick_sort_sub()首先决定 pivot 的值，然后把数组的元素分成大于 pivot 和小于 pivot 两个类别*。

* 尽管这里的算法占了整篇代码的一大半，但为了不跑题，笔者对算法不进行详细的说明，还是请读者自己去解读。

然后，对于分类后还存在两个以上元素的数组，**递归地调用自身**（第 23 行和第 26 行）。

因为程序优先对数组的左侧进行处理，所以数组将会被排序成图 2-10 中的样子。

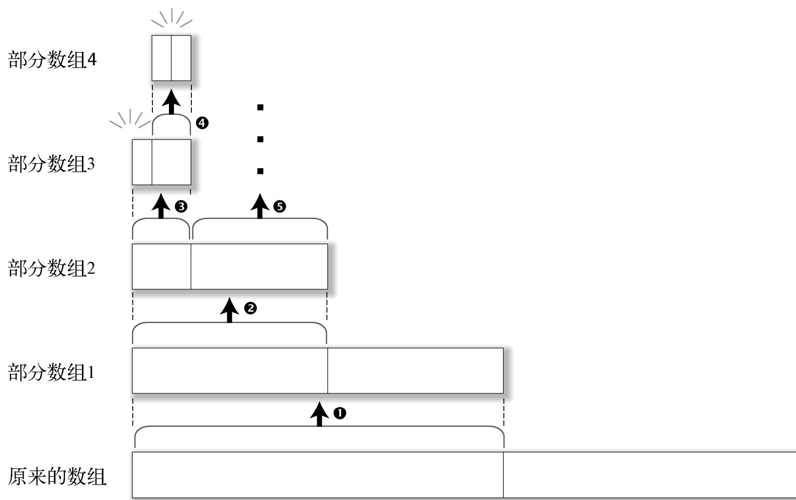


图 2-10　快速排序的概念图

首先，原始的数组被分类成大于 pivot 和小于 pivot 两个部分。对分类后的左侧（部分数组 1）递归调用自身再次进行分类（❶）。然后对分类的结果的左侧再分类，再分类……就这样不断地递归下去，一直进行到左侧还剩下一个元素（❹），再去处理右侧。就这样顺次地连续返回，对残留的部分进行处理（❺）。

那么在这里，处理完数组某部分的左侧之后，之所以处理还能够转移到右侧，是因为 C 将自动变量分配在栈中。

这个程序将数组分成两部分之后，再进行递归调用。此时，**在栈中分配了新的用于当前部分数组的内存区域**。另外，当某部分数组的数据处理完毕，函数调用返回的时候，栈会收缩，调用前的状态恰好处于栈的最初位置。正因为如此，处理才这样不断地向右移动。

如果这个程序不写成递归，程序写起来会变得有点麻烦。

拒绝偏见，请大家慢慢习惯去运用递归。



补充

ANSI C 以前的 C，为什么不能初始化自动变量的聚合类型？

ANSI C 以前的 C，只有标量才能在声明的同时被初始化（参照 1.1.8 节）。

因此，在如今能这样写的代码：

```
void func(void)
{
    int array[] = {1, 2, 3, 4, 5};
        ⋮
```

曾经只能写成下面这样：

```
void func(void)
{
    static int array[] = {1, 2, 3, 4, 5};
        ⋮
```

就像现在看到的这样，局部变量的内存区域是在函数被调用时，也就是**在执行中**被分配的。在执行中，如果想要初始化数组等聚合类型，编译

器必须在现场生成一些代码，编译器就被复杂化。你应该还记得前面提到过“原本 C 是只能使用标量的语言”吧。

对于这这种情况，如果加上 static 作为静态变量分配内存区域，就可以在程序的**执行前**被完全初始化。

从 ANSI C 开始，聚合类型的自动变量也可以在声明的时候进行初始化。可是，尽管这样还是会花费相应的成本，在可以使用静态数组的情况下，如果添加 static 可以获得令人满意的运行效率。如果同时加上 const，效率可能会更好。

此外，就算是 ANSI C，在聚合类型的初始化运算符中也只能写常量。

比如，

```
void func(double angle)
{
    double hoge = sin(angle);
          :
```

这样写是没有问题的，但

```
void func(double angle)
{
    double hoge[] = {sin(angle), cos(angle)};
          :
```

这么写就违反语法规范了。

Rationale 的 3.5.7 中说明了理由。下面的代码似乎会引起理解上的混乱，

```
int x[2] = { f(x[1]), g(x[0]) };
```

由于聚合类型的初始化运算符中只能写常量，所以编译器的处理会变得简单一些。因为只要事先在某个地方分配一个填充初始值的内存区域，然后在进入程序块的时候复制这个区域就可以了。

2.6　利用 malloc()来进行动态内存分配（堆）

2.6.1　malloc()的基础

C 语言中可以使用 malloc()进行动态内存分配。

malloc()根据参数指定的尺寸来分配内存块，它返回指向内存块初始位置的指针，经常被用于动态分配结构体的内存领域、分配执行前还不知道大

小的数组的内存领域等。

```
p = malloc(size);
```

一旦内存分配失败（内存不足），malloc()将返回 NULL。利用 malloc()分配的内存被结束使用的时候，通过 free()来释放内存。

```
free(p);  ←—释放 p 指向的内存区域
```

以上是 malloc()的基本使用方式。

像这样能够动态地（运行时）进行内存分配，并且可以通过任意的顺序释放的记忆区域，称为**堆**（heap）*。

* 它不是 C 语言定义的术语。

英语中“heap”这个单词是指像山一样堆得高高的事物（比如干草等）。malloc()就好像从内存的山上分出一部分内存，为我们“从堆中取来所需的内存区域”。

malloc()主要有以下的使用范例：

❶ 动态分配结构体

藏书家可能会用计算机管理自己的藏书。刚从书店买回来一本书，却发现：“啊！原来这本书我已经有了！”特别是漫画书，经常会有重复购买的现象，有吧？（难道只有我是这样？）

因此，最好做一个“藏书管理程序”。

假设用下面这样的结构体管理一本书的数据：

```
typedef struct {
    char  title[64];  /*书名*/
    int   price;      /*价格*/
    char  isbn[32];   /*ISBN*/
          ⋮
    } BookData;
```

对于藏书家来说，他一定要管理堆积如山的 BookData。

在这种情况下，当然可以通过一个巨大的数组来管理大量的 BookData，但是在 C 中必须明确定义数组的长度，究竟需要定义多大的数组是一件让人头疼的事情。如果申请过大的数据空间浪费内存，但如果申请的大小刚刚好，随着书的增加，数组的空间又会变得不足。

通过下面的方式，就可以在运行时分配 BookData 的内存区域。

```
BookData *book_data_p;

/*分配一个结构体 BookData 的内存区域*/
book_data_p = malloc(sizeof(BookData));
```

如果使用**链表**来管理，就可以持有任意个数的 BookData。当然，只要你内存足够。

关于链表的使用方法，在这里只是稍稍提一下，在第 5 章会详细说明。

首先让我们向结构体 BookData 中追加一个指向 BookData 类型的指针成员。

```
typedef struct BookData_tag {
    char  title[64];  /*书名*/
    int   price;      /*价格*/
    char  isbn[32];   /* ISBN */
      :
    struct BookData_tag *next;    ←—追加这一行
} BookData;
```

在这个例子中，利用 typedef 为 struct BookData_tag 定义同义词 BookData（不用一次次地写 struct 了）。尽管如此，由于成员 next 声明时 typedef 还没有结束，所以请注意这里必须要写成 struct BookData_tag。tag 不能省略。

通过 next 持有“指向下一个 BookData 的指针”，像图 2-11 这样形成串珠的数据结构，就能够保存大量的 BookData*。

* 以后，图中的●→表示指针，☒表示 NULL。

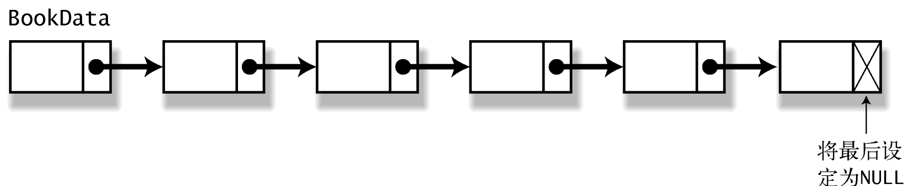


图 2-11　链表

这种被称为“链表”的数据结构运用非常广泛。

❷ 分配可变长数组

在刚才的 BookData 类型中，书名的地方写成下面这样：

```
char  title[64];  /*书名*/
```

可是有的情况下，书名也很变态，比如，

京都露天混浴澡堂蒸气谋杀事件——美女大学生 3 人组所见！
谜一样的旅馆女主人死亡和 10 年前的观光客失踪事件的背后所潜藏的真相是？

（这还是书名吗？）

此时，char title[64]就放不下这么长的书名了。可是，也不是所有的书都有这么长的书名，所以准备太长的数组也是浪费。

这里，将 title 的声明写成：

```
char  *title;  /*书名*/
```

然后，

```
BookData *book_data_p;
    ⋮
/*这里 len 为标题字符数+1（空字符部分的长度） */
book_data_p->title = malloc(sizeof(char) * len);
```

就能够只给标题字符串分配它必要的内存区域了。

此时，如果想要引用 title 中某个特定的字符，当然可以写成

```
book_data_p->title[i]
```

大家想必还记得 p[i]是*(p + i)的语法糖吧。



补充　malloc()的返回值的类型为 void*

ANSI C 以前的 C，因为没有 void*这样的类型，所以 malloc()返回值的类型就被简单地定义成 char*。char*是不能被赋给指向其他类型的指针变量的，因此在使用 malloc()的时候，必须要像下面这样将返回值进行强制转型：

```
book_data_p = (BookData*)malloc(sizeof(BookData));
```

ANSI C 中，malloc()的返回值类型为 void*，void*类型的指针可以不强制转型地赋给所有的指针类型变量。因此，像上面的强制转型现在已经不需要了。

尽管如此，现在还有人时常在这里使用强制转型。我认为，不写多余的强制转型代码，对于顺畅地读懂代码是有利的。

此外，假设忘记了#include stdlib.h，一旦笨拙地对返回值进行强制转型，编译器很可能不会输出警告。

C 语言默认地将没有声明的函数的返回值解释成 int 类型*，那些运气好、目前还能跑起来的程序，如果被迁移到 int 和指针长度不同的处理环境中，就会突然跑不动了。

* 如果可能，应该提高警告的级别，以便让警告可以输出。

因此，我们不要对 malloc()的返回值进行强制转型。因为 C 不是 C++。

另外，C++中可以将任意的指针赋给 void*类型的变量，但不可以将 void*类型的值赋给通常的指针变量。所以在 C++中，malloc()的返回值必须要进行强制转型。但是，如果是 C++，通常使用 new 来进行动态内存分配（也应该这样）。

2.6.2 malloc()是“系统调用”吗

这里有一点离题。

C 的函数库中为我们准备了很多的函数（printf()等）。另外，标准库的一部分函数最终会调用“系统调用”。所谓**系统调用***，就是请求操作系统来帮我们做一些特殊的函数群。标准函数通过 ANSI C 进行了标准化，但是不同操作系统上的系统调用的行为却经常会有差别。

* 这原本是 UNIX 的术语。

比如在 UNIX 操作系统中，printf()最终调用 write()这样的系统调用。不只是 printf()，putchar()和 puts()在最终也是调用 write()。

由于 write()只能输出指定的字节串，为了让应用开发人员更方便地使用和提高可移植性，C 语言给 write()披上了标准库的“皮”。

话说回来，malloc()是系统调用？还是标准库函数？

可能很多人会认为它是系统调用。恰恰相反，malloc()实际上属于标准库函数，它不是系统调用。

> 要 点
> **malloc()不是系统调用。**

2.6.3 malloc()中发生了什么

malloc()大体的实现是，从操作系统一次性地取得比较大的内存，然后将这些内存“零售”给应用程序。

根据操作系统的不同，从操作系统取得内存的手段也是不一样的，在 UNIX 的情况下使用 brk()*的系统调用*。

* break 的略称。

* 在最近的实现中，有时也会使用 mmap()（后面会提到）。使用 malloc() 分配小区域的时候，还是使用 brk()。

请大家回想一下，在图 2-3 中，“利用 malloc 分配的内存区域”的下面，是一片范围很大的空间。系统调用 brk()就是通过设定这个内存区域的末尾地址，来伸缩内存空间的函数的。

调用函数的时候，栈会向地址较小的一方伸长。多次调用 malloc()时，会调用一次 brk()，内存区域会向地址较大的一方伸长。

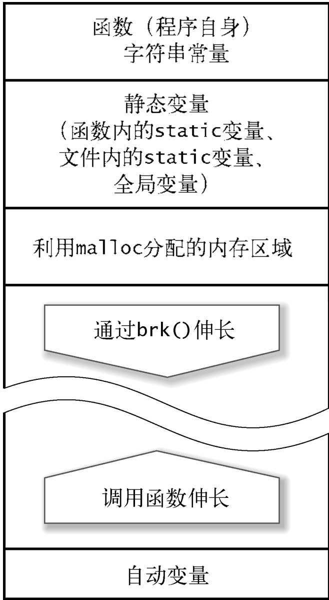


图 2-12 内存区域的伸长

“啥？就算可以通过这种方式分配内存区域，但是做不到以任意的顺序释放内存吧？”

也许会有很多人存在以上的想法。**实际上，这是有道理的。**

如果这么说，大家自然会想问：“那么 free()又是什么？”那么下面来说说 malloc()和 free()的基本原理。

现实中的 malloc()的实现，在改善运行效率上下了很多工夫。这里我们只考虑最单纯的实现方式——通过链表实现。

顺便提一下，*K&R* 中也记载了通过链表实现 malloc()的例程。

最朴素的实现就是如图 2-13，在各个块之前加上一个管理区域，通过管理区域构建一个链表。

malloc()遍历链表寻找空的块，如果发现尺寸大小能够满足使用的块，就分割出来将其变成使用中的块，并且向应用程序返回紧邻管理区域的后面区域的地址。free()将管理区域的标记改写成“空块”，顺便也将上下空的块合并成一个块。这样可以防止块的碎片化*。

* 这个操作被称为合并（coalescing）。也有不实现这个操作的，但这样会加剧碎片化（参照 2.6.5 节）。

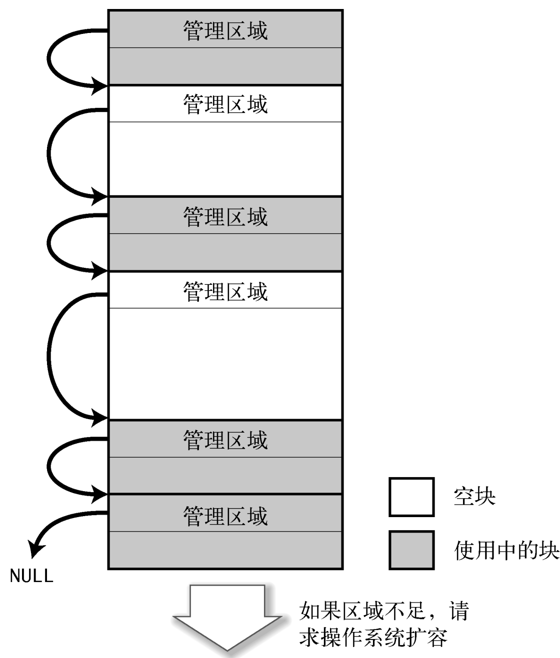


图 2-13　通过链表实现 malloc()的例子

如果不存在足够大的空块，就请求操作系统对空间进行扩容（UNIX 下使用 brk()系统调用）。

那么，在这种内存管理方式的运行环境中，一旦数组越界检查发生错误，越过 malloc()分配的内存区域写入了数据，又会发生什么呢？

此刻将会破坏下一个块的管理区域，所以从此以后的 malloc()和 free()

调用中出现程序崩溃的几率会非常高。这种情况下，不能因为程序是从 malloc()中崩溃的就一口咬定“这是库的 bug!!!”这么做只会自取其辱。

现实中的处理环境，是不会这样单纯地实现 malloc()功能的。

比如，作为内存管理方法，除了这里说明的链表方式之外，还有一个被大家广泛熟知的“buddy block system”方法。这种将大的内存逐步对半分开的方式，虽然速度很快，但会造成内存的使用效率低下。

此外，让管理区域和传递给应用程序的区域相邻也是比较危险的，所以有的实现中会将它们分开存放。我的环境（FreeBSD3.2）就是这样的（现行的实现中也有很多是相邻存放的）。

不过，我在这里只是想说明“malloc()绝对不是一个魔法函数”。

随着 CPU 和操作系统的不断进化，也许有一天 malloc()会成为真正的魔法函数。但目前你肯定不能将 malloc()看成魔法函数。对于 malloc()的工作原理，如果不是非常了解，就很有可能陷入程序不能正常进行调试的窘境，或者常常写出非常低效的程序。

所以我建议大家先充分理解 malloc()后再去使用它，不然它只会给你带来危险。

> 要　点
> **malloc()绝对不是魔法函数。**

2.6.4　free()之后，对应的内存区域会怎样

正如刚才所说，malloc()管理从操作系统一次性地被分配的内存，然后零售给应用程序，这是它大致的实现方式。

因此，一般来说调用 free()之后，**对应的内存区域是不会立刻返还给操作系统的**。让我们通过代码清单 2-7 来做个实验。

代码清单 2-7　free.c

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  int main(void)
5:  {
6:      int *int_p;
7:
```

```
 8:     int_p = malloc(sizeof(int));    ←在利用malloc()分配的
                                          内存区域中……
 9:
10:     *int_p = 12345;    ←写入
11:
12:     free(int_p);    ←调用free()之后
13:
14:     printf("*int_p..%d\n", *int_p);    ←输出对应的内容!
15:
16:     return 0;
17: }
```

在我的环境中执行这个程序，会输出12345。之后随着某次 malloc()调用，恰好将这片区域重新进行分配后，才会发生这部分内容的改写。

但是，C标准不能保证情况总是这样。因此，调用 free()之后，**是不能引用对应的内存区域的**。

这里之所以特地举出这个例子，是因为“调用 free()之后，对应的内存内容不会被马上破坏”。这样的特性给程序调试中的原因查明带来了困难。

如图2-14，某内存区域被两个指针同时引用。使用指针A引用这个区域的程序员认为当前区域对他来说已经不需要了，于是稀里糊涂地调用了 free()。实际上，在远离当前这段代码的地方，指针B还在引用当前这片区域。此时，会发生什么呢*?

* 大型的程序常常会出现这种问题。

仓促地调用 free()是有问题的，就算调用了 free()，指针B引用的内存区域也不会立刻被破坏，暂时还保持着以前的值。直到在某个地方执行 malloc()，随着当前内存区域被重新分配，内容才开始被破坏。这样的bug，从原因产生到bug被发现之间周期比较长，因此给程序调试带来很大困难。

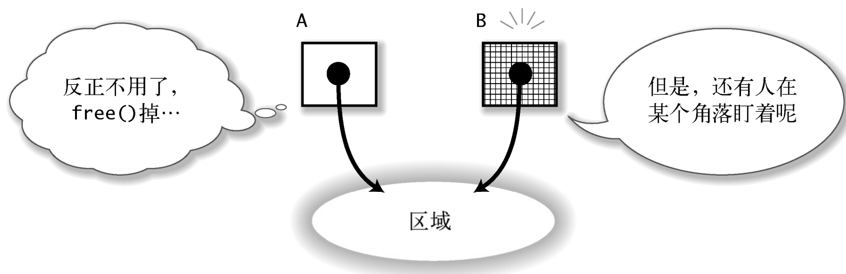


图2-14　一个内存区域被两个指针引用……

为了避免这个问题，如果是大型程序，可以做一个函数给 free()披一张皮，并且使程序员们只能调用这个函数，在区域被释放之前故意将区域破坏

（可以胡乱地填充一个像 0xCC 这样的值）。可惜的是，我们无法知道当前指针指向的区域的大小*。如果偏要这么做，可以考虑也给 malloc()披上一张皮，每次分配内存的时候可以多留出一点空间，然后在最前面的部分设定区域的大小信息。

* 如果使用 malloc()分配内存，标准库肯定是知道此内存的大小的。遗憾的是，ANSI C 没有提供公开内存大小的函数。

当然，这种手法只能用于程序的调试版本。在去掉调试选项对程序进行编译的时候，这些代码就会消失。这样就不会影响发行版的程序的运行效率。

这么做虽然有点麻烦，但是对于大规模的程序来说，这种手法还是非常有效的。

2.6.5 碎片化

某些处理环境对 malloc()的实现，和 2.6.3 节中描述的没有大的差别，但是以随机的顺序分配、释放内存。此时，又会发生什么问题呢？

此时，内存被零零碎碎分割，会出现很多细碎的空块。并且，这些区域**事实上是无法被利用的**。

这种现象，我们称为碎片化（fragmentation）（参照图 2-15）。

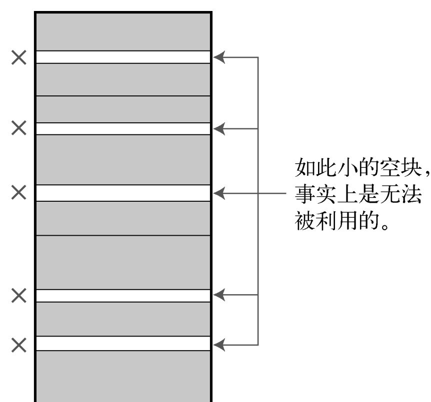


图 2-15 碎片化

将块向前方移动，缩小块之间的距离，倒是可以整合零碎的区域并将它们组合成较大的块*。可是 C 语言将虚拟地址直接交给了应用程序，库的一方是不能随意移动内存区域的。

* 这样的操作称为压缩（compaction）。

在 C 中，只要使用 malloc()的内存管理过程，就无法根本回避碎片化问题。但若利用 realloc()函数（在下一节说明），倒可以让问题得到一些改善。

2.6.6　malloc()以外的动态内存分配函数

本书介绍 malloc()以外的动态内存分配函数，首先介绍 calloc()函数。

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size)
```

calloc()使用和 malloc()相同的方式分配 nmemb × size 大小的内存区域，并且将该区域清零返回。也就是说，calloc()和下面的代码具有同等效果。

```
p = malloc(nmemb * size);
memset(p, 0, nmemb * size);
```

老实说，我对 C 语言提供这个函数的意图**完全无法理解**。通过两个参数传递区域的尺寸，实际上只是在内部做乘法运算，这点就让人费解。将区域清零返回这一点也让人感觉别扭。因为就算通过 memset()清零，在浮点数和指针的情况下它们的值也不一定为 0（也许是空指针）*。

* 可是，在现行的大部分处理环境中，浮点数的值被初始化为浮点数 0，指针被初始化为空指针。这无疑是多此一举地将问题复杂化。

calloc()对于避免难以再现的 bug 是有效的。我一般是给 malloc()披一张皮，但不是清零，而是填充一个无意义的值 0xCC。对于在那些没有好好做初始化的程序中发现 bug，我的方式肯定更好一些。

此外，在 *K&R* 的第一版中有一个 cfree()函数*，看上去好像可以利用这个函数释放由 calloc()分配的内存。其实这个函数和 free()做的事完全一样。现在，为了维持向后的兼容性，在大部分的环境中仍然可以使用 cfree()。但是 **ANSI C 的标准中是不存在这个函数的**。就算你使用了 calloc()，在释放内存的时候也请使用 free()。

* 随便提一下，实际上在 *K&R* 的第 1 版中，只记载了 calloc() 和 cfree()的配对函数，并**没有记载** malloc()。

> 要　点
>
> **不要使用 cfree()。**

还有一个内存分配函数是 realloc()。

此函数用于改变已经通过 malloc()分配的内存的尺寸。

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

realloc()将 ptr 指定的区域的尺寸修改成 size，并且返回指向新的内存区域的指针。

话虽这么说，但正如前面说明的那样，malloc()不是一个魔法函数，

realloc()同样如此。通常，我们使用 realloc()来扩展内存区域。如果通过 ptr 传递的区域后面有足够大小的空闲空间，就直接实施内存区域扩展。但是，如果后面的区域没有足够多的空闲空间，就分配其他新的空间，然后将内容复制过去。

我们经常需要对数组顺次追加元素，这种情况下，如果每追加一个元素都利用 realloc()进行内存区域扩展，将会发生什么呢？

如果手气不错，后面正好有足够大的空地儿，这样还好。如果不是这样，就需要频繁地复制区域中的内容，这样自然会影响运行效率。另外，不断地对内存进行分配、释放的操作，也会引起内存碎片化。此时不妨考虑一下这种手法：假设以 100 个元素为单位，一旦发现空间不足，就一次性进行内存扩展分配*。

* Java 的 java.util.Vector 类就是采用了这个手法。但是本质上，Java 并不背离堆的机制。

此外，一旦利用 realloc()扩展巨大的内存区域，除了在复制上花费很多时间之外，也会造成堆中大量的空间过分地活跃。如果想要动态地为大量的元素分配内存空间，最好不要使用连续的内存区域，而是应该积极使用链表。

> 要　点
> **请谨慎使用 realloc()。**

另外，如果通过 ptr 向 realloc()传入 NULL，realloc()的行为就和 malloc()完全相同。偶尔会见到像下面这样的代码，

```
if (p == NULL) {
    p = malloc(size);
} else {
    p = realloc(p, size);
}
```

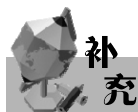
完全可以简洁地写成下面这样，

```
p = realloc(p, size);
```

（姑且先把返回 NULL 的问题放在一边*）。

* 对于这种写法，realloc()返回 NULL 的时候，p 会永远被丢失，这是一个问题。

随便说一下，如果通过 size 向 realloc()传入 0，realloc()的行为就和 free()完全相同。



补充　malloc()的返回值检查

内存分配一旦失败，malloc()会返回 NULL。

大部分 C 的书籍都歇斯底里地提醒大家“如果你调用了 malloc()，就**必须**做返回值检查”，关于这一点，本书倒是要对此唱唱反调。不是吗？那么做**太麻烦了**。

完美地对应内存不足的问题，

```
p = malloc(size);
if (p == NULL) {
    return OUT_OF_MEMORY_ERROR;
}
```

并不是像上面这样机械地写一段代码那么单纯的。

当然我们在构造某些数据结构的时候，既需要确保数据结构自身没有任何矛盾，还要确保函数能够返回，随之而来的测试也会变得更加复杂。假设能够做到这些，却又因为分配已达上限字节数的内存区域而失败，此时我们又该怎么办呢？

- 通过对话框通知用户“内存不足”？此时，也许程序自身已经没有能力弹出对话框了……
- 为了不丢失写到一半的文档数据，姑且将文件保持打开状态……能做到吗？
- 安全起见，通过递归的方式遍历深层次的树结构……可以分配栈的空间吗？
- 还是先保护硬盘数据吧……对于 Windows 这样存放了交换文件的文件系统，如果此时恰恰只有一个分区，硬盘数据又会出现什么状况呢？

其实并不是只有显式的 malloc()调用才会导致内存不足，深度的递归调用也会引起栈内存不足。此外，根据不同情况，printf()内部有时也会调用 malloc()。不光是 malloc()调用，操作系统往内存中写入数据的时候也会进行内存区域分配，对于这种情况，调用 malloc()时是无法发现内存不足的。

开发通用性极高的库程序的时候，“切实地进行返回值检查”的确是必须的。但是如果是开发普通的应用程序，只需要给 malloc()披上一张皮，一旦发生内存不足，当场输出错误信息并且终止该程序，这种做法在很多时候也是可行的。主张

> “一旦调用 malloc()，绝对要对返回值进行检查，并且进行完善的处理。”

的人，一般在使用 Java 时，一定会在合适的层次上使用 catch，当然他们也不会去使用 Perl 等等的 Shell 脚本……



补充 程序结束时必须调用 free()吗？

网上的新闻组 fj.comp.lang.c 曾经针对下面的主题发生过一次激烈的口水战。

> 程序结束之前，一定需要释放 malloc()分配的内存吗？

这是一个让人头疼的问题。在现实中，如果是普通 PC 上使用的操作系统，在进程结束时，肯定会释放曾经分配给当前进程的内存空间。

其实 C 语言标准却并没有规定必须要这么做，只是正经的操作系统都主动提供这个功能。此外，在写推荐内存为 128MB 的程序时，你不会去考虑以后还要将它移植到电视机遥控器的嵌入式芯片上吧。也就是说，在程序结束之前，没有必要调用 free()。

可是，对于进行“读取一个文件→处理→输出结果→结束”这样的处理，如果要扩展成可以连续处理多个文件，一旦原来的程序没有调用 free()，后面的人那可真的遭罪了。

为了提前发现内存溢出（忘了调用 free()）的漏洞，最近出现了一些工具，它可以报告那些没有在结束时被实施 free()的内存区域的列表。此时，“故意不调用 free()的区域”和“忘记调用 free()区域”被混同在一起出现在报告中，让人难以区分。对于不能使用这些工具的开发环境，可以采用“将 malloc()和 free()披上一张皮，然后计算它们被调用的次数，并且确认程序结束时次数是否一致”这样简单可行的方法，并且这种方法对于检查内存泄漏非常有效。

从这一点上来看，我认为“对于调用 malloc()分配的内存区域，在程序结束前一定要调用 free()”这样的原则也是相当合理的。

那么到底应该怎么做？答案是“具体问题具体分析”（我倒！跟没说一样）。

我倒是不太喜欢“必定 free()派”的观点，其实之所以“必定 free()”，是因为他们认为：

- 使用 malloc()之后写上对应的 free()是一种**谨慎的编程风格**
- 程序员就应该**留意**将 malloc()和 free()对应起来
- “调用了 exit()，就没有必要调用 free()了”这种想法是**不负责任并且恶劣的**编程风格

不管怎么说程序员也是人（瞧这话说的），对于人来说，恐怕会犯错的地方必定犯错。明明如此，你还去标榜什么“写程序要谨慎”，我觉得有点自讨没趣。

"谨慎地"编程有那么了不起吗？我认为那些能尽力让自己摆脱"麻烦事"的程序员才是优秀的。该脱手时就脱手，尽可能依赖工具去完成检查工作而不是总去目测。就算在那些无论如何也要依靠手动去应对的情况下，也暗自发誓"总有一天我把它做成自动化"。此类程序员才是人才！

2.7　内存布局对齐

稍微转换一下话题……

假设有下面这样的一个结构体：

```
typedef struct {
    int     int1;
    double  double1;
    char    char1;
    double  double2;
} Hoge;
```

在我的环境中，sizeof(int)的结果为 4，sizeof(double)的结果为 8，随便说一下，根据 C 标准，sizeof(char)的结果必定为 1*。敢问阁下，这个结构体的尺寸是多大？

* 比如，即使 char 为 9 bit 的处理环境（如果真的存在），sizeof(char) 值也是 1。标准就是这么定义的。

4 + 8 + 1 + 8 = 21 个字节——几乎在所有的情况下，这个答案都是错误的。在我的处理环境中，答案是 24 个字节。

还是通过程序做个实验吧（参照代码清单 2-8）。

声明一个 Hoge 类型的变量，然后将各成员的地址输出*。

* 如果需要获得结构体成员距离初始位置的偏移量，一般使用 stddef.h 中定义的宏 offsetof()。使用这个宏，不需要声明哑变量（dummy）也可以获取偏移量。

代码清单 2-8　alignment.c

```
 1:  #include <stdio.h>
 2:
 3:  typedef struct {
 4:      int     int1;
 5:      double  double1;
 6:      char    char1;
 7:      double  double2;
 8:  } Hoge;
 9:
10:  int main(void)
11:  {
12:      Hoge        hoge;
13:
14:      printf("hoge size..%d\n", sizeof(Hoge));
```

```
15:
16:     printf("hoge   ..%p\n", &hoge);
17:     printf("int1   ..%p\n", &hoge.int1);
18:     printf("double1..%p\n", &hoge.double1);
19:     printf("char1  ..%p\n", &hoge.char1);
20:     printf("double2..%p\n", &hoge.double2);
21:
22:     return 0;
23: }
```

我的环境中的运行结果如下：

```
hoge size..24
hoge   ..0xbfbfd9d0
int1   ..0xbfbfd9d0
double1..0xbfbfd9d4
char1  ..0xbfbfd9dc
double2..0xbfbfd9e0
```

观察运行结果可以发现，char1 的后面空出来一块。

这是因为根据硬件（CPU）的特征，对于不同数据类型的可配置地址受到一定限制。或者，即使可以配置，某些 CPU 的效率也会降低。此时，编译器会适当地进行边界调整（布局对齐），在结构体内插入合适的填充物。

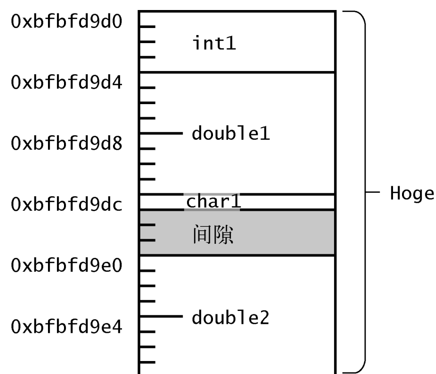


图 2-16　布局对齐

根据这个实验，在我的环境中，int 和 double 被配置在 4 的倍数的地址上。

布局对齐处理有时候也在结构体的末尾进行，这是由于有时候需要构造结构体数组的缘故。针对这样的结构使用 sizeof 运算符，会返回包含末尾对

齐字节的结构体长度。将结果和元素个数相乘，就可以获得整个数组的大小。

此外，malloc()会充分考虑到各种类型的长度，返回调整后最优化的地址。局部变量等也会被配置到优化调整后的地址上。

布局对齐操作是根据 CPU 的情况进行的。因此，**根据 CPU 的不同，布局对齐填充的方式也不同**。在我的环境中，double 可以被配置在 4 的倍数的地址上，但在很多 CPU 上，double 只能被配置在 8 的倍数的地址上。

偶尔，也会有人比较讨厌布局对齐方式对硬件的依赖，通过手工调整边界来提高可移值性。

```
typedef struct {
    int     int1;
    char    pad1[4];    ←—通过手工填充
    double  double1;
    char    char1;
    char    pad2[7];    ←—这里也是
    double  double2;
} Hoge;
```

可是，这么做究竟有什么作用呢?

即使不这么做,编译器也会根据 CPU 的情况帮我们进行适当的边界调整。如果只是引用成员名，就根本没有必要去理会布局对齐方式。

如果需要将结构体照原样（通过 fwrite()）输出到文件中，由于 CPU 的不同，在其他机器上想要读取这个结构体的时候，对齐方式的不同可能导致问题。那么，通过手工方式调整边界，说不定某台机器上输出的数据，也能被其他机器读取。可是无论怎样，这只不过是**偶尔**才可以拿出来说的例子。

在上面的例子中，pad1 的尺寸为 4，pad2 的尺寸为 7。**究竟这些数字是怎么冒出来呢**？连标准也不能保证 sizeof(int)为 4，sizeof(double)为 8。将这些数字直接写在程序中，还说什么“为了提高可移植性”……

也就是说，手工插入填充物的方法，即使可以让不同机器的数据交换成为可能，也只不过是敷衍逃避。原型开发也许会允许使用这种方式，但是如果在现实中进行数据交换，**将结构体按照原样写入到文件的方式本身就是个错误**。

另外，即使是 sizeof(int)为 4 的处理环境，其内部表现也不一定相同。关于这一点，下一节会进行说明。

要　点

即使手工进行布局对齐，也不能提高可移植性。

2.8　字节排序

我的环境是在普通 PC（CPU 为 Celeron）上安装了 FreeBSD3.2。sizeof(int)为 4，但在这 4 个字节中，整数究竟是以什么样的形式存放的呢？

这里依然使用一个程序来验证问题（代码清单 2-9）。

代码清单 2-9　byteorder.c

```
 1:  #include <stdio.h>
 2:
 3:  int main(void)
 4:  {
 5:      int           hoge = 0x12345678;
 6:      unsigned char *hoge_p = (unsigned char*)&hoge;
 7:
 8:      printf("%x\n", hoge_p[0]);
 9:      printf("%x\n", hoge_p[1]);
10:      printf("%x\n", hoge_p[2]);
11:      printf("%x\n", hoge_p[3]);
12:
13:      return 0;
14:  }
```

程序中将 int 型变量的地址强制赋给 unsigned char *型变量 hoge_p，因此，我们可以使用 hoge_p[0]~hoge_p[3] 以字节为单位引用 hoge 的内容。

我的环境中，程序的执行结果如下：

```
78
56
34
12
```

对于我的环境，“0x12345678”在内存中好像是逆向存放的呢。

可能有读者会感到意外。其实 Intel 的 CPU（包括 AMD 等兼容 CPU），

都是像这样将整数**颠倒过来**存放的。这种配置方式一般称为小端(little-endian)字节序。

此外，对于工作站等的 CPU，经常将“0x12345678”这样的值以“12，34，56，78”的顺序存放，这种配置方式称为大端（big-endian）字节序。

那么，小端和大端这样的字节排列方式就称为字节排序（Byte Order）。

小端和大端哪一种方式会更好？这个问题如同辩论宗教取向一样令人纠结，在这里我们就不做深入讨论了。事实是它们各有所长。在纸和笔的时代，人类在做加法运算时都是从低位开始的，可是对于 CPU 来说采用小端方式会更轻松。当然，对于人类来说，大端的方式也许更容易理解。

问题就在于，连以上这样整数类型的数据**在内存中的配置方式都因 CPU 的不同而不同**。

事实上，有的 CPU 还会采用“将两个字节编成一组再反向排列”这样差异性更大的字节排列方式。此外，很多环境使用 IEEE 754 规定的形式处理浮点类型，但 C 的标准并没有规定这一点*。即使是采用 IEEE 754 的处理环境，Intel 的 CPU 也还是逆向排列字节的。

* Java 也规定了这一点。因此，如果硬件不支持 IEEE 754，情况就会变得复杂一点。

也就是说，根据环境的不同，内存中的二进制映像的形式也不尽相同，所以那些**试图将内存的内容直接输出到硬盘，或者通过网络进行传输以便不同的机器读取等想法都是不可取的**。

如果要考虑数据兼容性，建议自定义一些数据格式，然后遵循这些格式来输出数据。UNIX 的 XDR 等工具可以在这一点上为我们提供帮助。

> 要　点
> **无论是整数还是浮点小数，内存上的表现形式都随环境的不同而不同。**

2.9　关于开发语言的标准和实现——对不起，前面的内容都是忽悠的

直到这里，都是在我的环境中实际地运行程序，然后结合输出结果进行各种各样的说明。

但是，C 标准并不是根据实现方式制定的，而是根据对开发语言的需求来制定的。

比如,如今大多的PC和工作站的操作系统都为我们实现了虚拟地址功能，但是对于那些不具备此功能的操作系统，C 语言编写的应用程序同样能跑得很欢。

此外，前面介绍了“C 语言将自动变量分配在栈中”。其实，这一点并没有在标准中做出规定。因此，即使在每次进入函数的时候将自动变量配置到堆中，也不算违反标准。只是这种实现非常慢，没有人会愚蠢到这种地步罢了。

至于 malloc()的实现，在不同的处理环境中也存在很大的差异。除了 brk()这样 UNIX 特有的系统调用之外，最近也出现了这样的分配方式：在分配较大的内存空间时，使用系统调用 mmap()，之后通过 free()将内存返回给操作系统*。

* 默认地不一定被关联到那里。

进一步说，第 1 章的内容都是以“指针就是地址”为前提进行说明的。可是关于这一点，**在标准中连一个字也没有提到**。正如前面摘录的内容一样，标准中只提到“指针类型描述一个对象，该类对象的值提供对该引用类型实体的引用”。也就是说，如果连实体都能够被引用，即使你不使用虚拟地址也不违反标准。

在标准中，&运算符被称为“地址运算符”，这就让人有点糊涂，具体的说明是：

> 一元&（地址）运算符的结果是指向由其操作数所表示对象或函数的指针。

无论怎样，此运算符返回的是“指针”，而不是“地址”。

通过%p，printf()的输出结果被定义成：

> 该指针的值将以实现定义的方式转换为一系列可打印的字符。

由于 C 语言经常通过指针的形式让我们可以直接接触到地址，所以人们常常误以为：

- 如果没有养成经常关注内容状态的习惯，是不适合进行 C 语言编程的
- C 语言其实就是结构化的汇编
- C 语言其实是低级语言

但对于开发应用程序的普通程序员，其实**完全没有必要去理会指针就是地址**这件事。

可能 C 语言确实就是低级语言。也许有人想说：

> 明明想使用高级语言，只是在不得已用了 C 语言的情况下，还装腔作势刻意地来一句“啥？C 难道不是低级语言吗”，并以此来标榜自己能够使用低级语言编程，这样可以让他人高看自己一眼。

其实不是这样的。你明知处理环境中指针就是地址，却对此**选择性失明**，并且坚持抱着“如果将 C 当成高级语言，C 就可以像高级语言那样使用”的想法。那么，等待你的将是成群的 bug。

本章也是在强调“指针就是地址”的基础之上，展开各部分内容的。

比起唠唠叨叨地进行那些抽象的说明，具体地将地址表现出来的方式是不是更直观？将自动变量的地址表示出来，你一下子就能看到栈是如何随着函数调用的发生而不断增长的。如果你没有理解这一点，你同样也无法理解递归调用的原理。

此外，在大部分的运行环境中，C 语言不做运行时检查，这已经是无法回避的现实。因此，如果没有在某种程度上掌握内存的使用方式，正常的调试工作就会遇到麻烦。

另外，遇到不明白的地方，应该通过实验来确认。俗话说得好，实践出真知！

对于本章的例程，请大家一定亲手在自己的环境中尝试运行一下。

但是，一旦大家通过实验接受了“指针就是地址”这个观点，就应该对此提高警惕，否则你就会顺手写出一些抽象度和移植度低下的程序。

此外，一旦出现 bug，请带着“指针就是地址”的观点去解决它——这种姿态在解决 bug 上是恰到好处的。

第 3 章

揭秘 C 的语法——它到底是怎么回事

3.1　解读 C 的声明

3.1.1　用英语来阅读

在 1.2.2 节的补充内容中，我认为像

```
int *hoge_p;
```

还有

```
int hoge[10];
```

这样的声明方式很奇怪。

对于这种程序的声明方式，可能也有很多人感觉不到有什么别扭的地方。那就再看下面的这个例子（经常被使用）：

```
char *color_name[] = {
    "red",
    "green",
    "blue",
};
```

这里声明了一个“指向 char 的指针的数组”。

正如 2.3.2 节中介绍的那样，可以像下面这样声明一个“指向将 double 作为参数并且返回 int 的函数的指针”：

```
int (*func_p)(double);
```

关于这样的声明，在 *K&R* 中有下面这样一段说明：

```
int *f();      /* f：返回指向 int 指针的函数*/
```

和

```
int (*pf)();   /* pf： 指向返回 int 的函数的指针*/
```

> 这两个声明最能说明问题。在这里，因为*是前置运算符，它的优先级低于()，为了让连接正确地进行，有必要加上括号。

首先，这段文字中**有谎言**。

声明中*、()和[]并不是运算符。在语法规则中，运算符的优先顺序是在别的地方定义的。

先将这个问题放在一边。如果你老老实实地去读这段文字，该会嘀咕“**是不是搞反了**”。如果说

```
int (*pf)();
```

是指向函数的指针，使用括弧先将星号（指针）括起来岂不是很奇怪？

这个问题的答案，等你明白过来就会觉得非常简单。C 语言本来是美国人开发的，**最好还是用英语来读**[*]。

* 在 *K&R* 中，登载了 dcl 这个解析 C 的声明的程序，同时也记载了程序的输出结果，但是日语版并没有对这一段进行翻译，而是一成不变地转载了英文原文。

以上的声明，如果从 pf 开始以英语的顺序来读，应该是下面这样：

```
pf is pointer to function returning int
```

翻译成中文，则为

```
pf 为指向返回 int 的函数的指针。
```

> 要　点
> **用英语来读 C 的声明。**

3.1.2　解读 C 的声明

在这里，向读者介绍阅读 C 语言声明的方法：机械地向前读。

为了把问题变得更简单，我们在这里不考虑 const 和 volatile。（3.4 节考虑了 const）接下来遵循以下步骤来解释 C 的声明。

❶ 首先着眼于标识符（变量名或者函数名）。
❷ 从距离标识符最近的地方开始，依照优先顺序解释派生类型（指针、数组和函数）。优先顺序说明如下，

①用于整理声明内容的括弧

②用于表示数组的[]，用于表示函数的()

③用于表示指针的*

❸ 解释完成派生类型，使用“of”、“to”、“returning”将它们连接起来。

❹ 最后，追加数据类型修饰符（在左边，int、double 等）。

❺ 英语不好的人，可以倒序用日语（或者中文）[①]解释。

数组元素个数和函数的参数属于类型的一部分。应该将它们作为附属于类型的属性进行解释。

比如，

```
int (*func_p)(double);
```

❶ 首先着眼于标识符。

```
int (*func_p)(double);
```

英语的表达为：

func_p is

❷ 因为存在括号，这里着眼于*。

```
int (*func_p)(double);
```

英语的表达为：

func_p is pointer to

❸ 解释用于函数的()，参数是 double。

```
int (*func_p)(double);
```

英语的表达为：

func_p is pointer to function(double) returning

❹ 最后，解释数据类型修饰符 int。

```
int (*func_p)(double);
```

英语的表达为：

func_p is pointer to function(double) returning int

① 这里同样可以倒序用中文解释。——译者注

❺ 翻译成中文：

func_p 是指向返回 int 的函数的指针。

使用和上面相同的方式，我们对各种各样的声明进行解读，如下表（表 3-1）。

表3-1　解读各种各样的C语言声明

C语言	英语表达	中文表达
`int hoge;`	hoge is int	hoge是int
`int hoge[10];`	hoge is array(元素个数10) of int	hoge是int的数组(元素个数10)
`int hoge[10][3];`	hoge is array(元素个数10) of array(元素个数3) of int	hoge是int数组(元素个数3)的数组(元素个数10)
`int *hoge[10];`	hoge is array(元素个数10) of pointer to int	hoge是指向int的指针的数组(元素个数10)
`double (*hoge)[3];`	hoge is pointer to array(元素个数3) of double	hoge是指向double的数组(元素个数3)的指针
`int func(int a);`	func is function(参数为int a) returning int	func是返回int的函数(参数是int a)
`int (*func_p)(int a) ;`	func_p is pointer to function(参数为int a) returning int	func_p是指向返回int的函数(参数为int a)的指针

正如大家看到的这样，C 语言的声明不能从左往右按顺序解读（无论是英语、中文，还是日语），而是左右来回地解读。

K&R 中指出：在 C 语言中，变量的声明仿效表达式的语法。可是，勉强地去模拟本质上完全不同的事物，结果就是“四不像”。

“使声明的形式和使用的形式相似”是 C（还有从 C 派生的 C++、Java*等语言）特有的**奇怪的语法**。

* 其实，Java 的大部分声明语法还是能做到这点的。

K&R 中同时也写道：

> C 的声明语法，特别是指向函数指针的语法，受到了严厉的批评。

在 Pascal 中，C 的 `int hoge[10]`可以这样声明，

```
var
    hoge : array[0..9] of integer;
```

这种声明，从左向右用英语按顺序解读是完全没有问题的。

顺便说一下，C 的作者 Dennis Ritchie 开发了一种叫 Limbo[7]的语言。Limbo 中各种标记的使用方法，一眼就可以看出来和 C 非常相似*，但是声明语法完全设计成 Pascal 风格。其实作者自身也在反省 C 语言的声明语法。

* 比如，不使用 begin～end 或者 if～endif 而是使用中括号。

3.1.3　类型名

在 C 中，除标识符以外，有时候还必须定义“类型”。

具体来说，遇到以下情况需定义“类型”：

- ❑ 在强制转型运算符中
- ❑ 类型作为 sizeof 运算符的操作数

比如，将强制转型运算符写成下面这样：

```
(int*)
```

这里指定“int*”为类型名。

从标识符的声明中，将标识符取出后，剩下的部分自然就是类型名。

表3-2　类型名的写法

声　明	声明的解释	类　型　名	类型名的解释
int hoge;	hoge是int	int	int类型
int *hoge;	hoge是指向int的指针	int *	指向int的指针类型
double (*p)[3];	p是指向double的数组（元素个数3）的指针	double(*)[3]	指向double的数组（元素个数3）的指针类型
void (*func)();	func是指向返回void函数的指针	void (*)()	指向返回void函数的指针类型

在表 3-2 最后两个例子中，括起星号的括弧(*)好像有点多余，**但是一旦去掉括弧，意思就完全不一样了**。

(double *[3])是将 double *hoge[3]的标识符去掉后形成的，所以这个类型名被解释成“指向 double 的指针的数组”。

补充　如果将指针后置……

C 的声明语法虽然奇怪，但也有人说 Pascal 风格写起来长得像裹脚布，同样让人感到厌恶。

```
var
    hoge : array[0..9] of integer;
```

关于这样的声明，array 后面紧接着[]，用来表示这是个数组，但是这样让人感觉 array 这个单词太长了。顺手在后面追加的 of 也是多余的。尝试将这些多余的部分去掉，结果就像下面这样：

```
var
    hoge : [0..9] integer;
```

如果改成 C 的写法：

```
hoge[10] int;
```

如果仅仅就 int 前置还是后置的问题来说，感觉和 C 的声明方式也没多大差别。

可是，一旦涉及指针，情况就不一样了。C 的指针运算符*是前置的。

在 Pascal 中，运算符^相当于 C 中*的，而且它是后置的。

如果同样地将 C 的指针运算符*也放在标识符后面，即使兼顾“变量的声明仿效表达式的语法”，声明也会变成下面这样：

```
int func_p^(double);
```

如果这个声明表示“指向返回 int 的函数（参数为 double）的指针”，差不多也符合英语的阅读顺序。不过 int 放在前面终究是个问题。

此外，一旦使用后置的^*，通过指针引用结构体的成员的时候，就可以不要->运算符了。

* C 中，^被作为异或运算符使用。这里，我们且不必去关心这一点。

原本

```
hoge->piyo
```

只是

```
(*hoge).piyo
```

的语法糖，所以又可以写成

```
hoge^.piyo
```

因此->完全可以不要的。

此外，将解引用后置，可以使包含结构体成员和数组引用的复杂表达式变得简洁*。

* 另外，如果将指针的强制转型也进行后置，同样也能起到简化表达式的作用。

关于这一点，“The Development of the C Language[5]”中也有说明：

Sethi [Sethi 81] observed that many of the nested declarations and

expressions would become simpler if the indirection operator had been taken as a postfix operator instead of prefix, but by then it was too late to change.

请允许我用我这二把刀的英语水平给大家翻译一下：

Sethi 认为，如果将解引用由前置变成后置，嵌套的声明和表达式就会变得更简单。但是，如今想要修正，为时已晚。

3.2　C 的数据类型的模型

3.2.1　基本类型和派生类型

假设有下面这样的声明：

```
int (*func_table[10])(int a);
```

根据上节中介绍的方法，可以解释成：

指向返回 int 的函数（参数为 int a）的指针的数组（元素个数 10）

如果画成图，可以用这样的链结构[①]来表示：

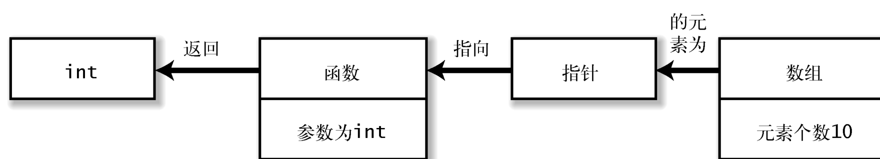


图 3-1　用图表现“类型”

这种表示，在本书中称为“类型链的表示”。

姑且先**忽视结构体、共用体、typedef 等类型**，概要地进行说明，链的最后面的元素*是**基本类型**，这里可能是 int 或者 double。

* 如果按照日语的语序，应该是最前面的元素。[②]

此外，从倒数第 2 个元素开始的元素都是**派生类型**。所谓“派生类型”，就是指从某些类型派生出来的类型。

① 由于日语和中文语序的差异，图中日文原书的箭头和本书相反。——译者注
② 译本采用中文的语序。——译者注

除了结构体、共用体之外，还有以下 3 种**派生类型**：

- 指针
- 数组（“元素个数”作为它的属性）
- 函数（参数信息作为它的属性）

关于派生类型，*K&R* 中有这样一段描述（p.239）：

> 除基本的算术类型以外，利用以下的方法可以生成概念上无限种类的派生类型。
>
> - 给出类型的对象的数组
> - 返回给出类型的对象的函数
> - 指向给出类型的对象的指针
> - 包含各种一系列对象的结构体
> - 能够包含各种类型的数个对象中的任意一个共用体
>
> 一般来说，这些对象的生成方法可以递归使用。

可能大家完全不明白这段描述在说什么。其实归纳一下，可以表述成下面这句话*，

* 实际上，派生还有其他几个限制，关于这些我们在后面介绍。

> 从基本类型开始，递归地（重复地）粘附上派生类型，就可以生成无限类型。

通过如图 3-1 的方式将链不断地接长，就可以不断生成新的“类型”。

另外，在链中，最后的类型（数组(元素 10)）整合了全体类型的意义，所以我们将最后的类型称为**类型分类**。

比如，无论是“指向 `int` 的指针”，还是“指向 `double` 的指针”，结果都是“指针”；无论是“`int` 的数组”，还是“指向 `char` 的指针的数组”，结果都是“数组”。

3.2.2　指针类型派生

在 1.2.1 节中，引用了标准中的一节，在此，请允许我再次引用。

> **指针类型**（pointer type）可由函数类型、对象类型或不完全的类型派生，派生指针类型的类型称为引用类型。指针类型描述一个对象，该类对象的值提供对该引用类型实体的引用。由引用类型 T 派

生的指针类型有时称为“(指向)T 的指针”。从引用类型构造指针类型的过程称为“指针类型的派生”。这些构造派生类型的方法可以递归地应用。

“由引用类型 T 派生的指针类型有时称为‘(指向)T 的指针’”这句话，可以在图 3-2 中用链来表现。

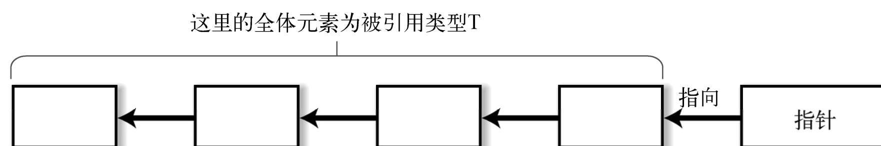


图 3-2 指针类型派生

对于指针类型来说，因为它指向的类型各不相同，所以都是从既存的类型派生出“指向 T 的指针”这样的类型。

大多数处理环境中的指针，在实现上都只是单纯的地址，所以无论从什么类型派生出的指针，在运行时的状态都是大体相同的*。但是，加上*运算符求值的时候，以及对指针进行加法运算的时候，由不同类型派生出来的指针之间就存在差异。

* 前面也曾经提到，如果解释得详细一些，对于指向 char 的指针和指向 int 的指针，偶尔存在位数不相同的处理环境。

前面已经提到，如果对指针进行加法运算，指针只前进**指针所指向类型的大小**的距离。这一点对于后面的说明有着非常重要的意义。

可以使用图 3-3 来解释指针类型。

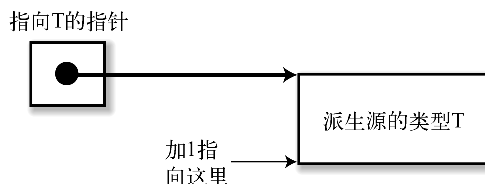


图 3-3 指针类型的图解

3.2.3 数组类型派生

和指针类型相同，数组类型也是从其元素的类型派生出来的。“元素个数”作为类型的属性添加在类型后面（参照图 3-4）。

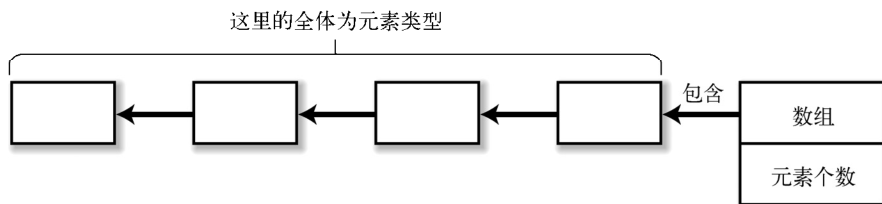


图 3-4 数组类型派生

数组类型本质就是将一定个数的派生源的类型进行排列而得到的类型。如图 3-5 所示。

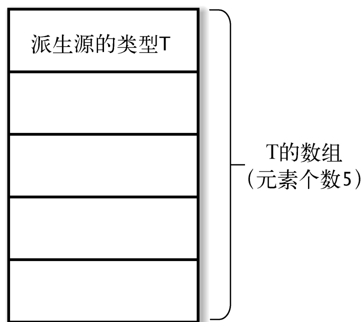


图 3-5 数组行的图解

3.2.4 什么是指向数组的指针

“数组”和“指针”都是派生类型。它们都是由基本类型开始重复派生生成的。

也就是说，派生出“数组”之后，再派生出“指针”，就可以生成“指向数组的指针”。

一听到“指向数组的指针”，有人也许要说：

> 这不是很简单嘛，数组名后不加[]，不就是“指向数组的指针”吗？

抱有这个想法的人，请将 1.3 节的内容重新阅读一下！的确，**在表达式中，数组可以被解读成指针**。但是，这不是“指向数组的指针”，而是“**指向数组初始元素的指针**”。

实际地声明一个“指向数组的指针”，

```
int (*array_p)[3];
array_p 是指向 int 数组（元素个数 3）的指针。
```

根据 ANSI C 的定义，在数组前加上&，可以取得“指向数组的指针”*。因此，

* 这里是“数组可以解读成指向它初始元素的指针”这个规则的一个例外（参照 3.3.3 节）。

```
int array[3];
int (*array_p)[3];

array_p = &array;     ←数组添加&，取得“指向数组的指针”
```

这样的赋值是没有问题的，因为类型相同。

可是，如果进行

```
array_p = array;
```

这样的赋值，编译器就会报出警告。

“指向 int 的指针”和“指向 int 的数组（元素个数 3）的指针”是完全不同的数据类型。

但是，从地址的角度来看，array 和&array 也许就是指向同一地址。但要说起它们的不同之处，那就是它们在做指针运算时结果不同。

在我的机器上，因为 int 类型的长度是 4 个字节，所以给“指向 int 的指针”加 1，指针前进 4 个字节。但对于“指向 int 的数组（元素个数 3）的指针”，这个指针指向的类型为“int 的数组（元素个数 3）”，当前数组的尺寸为 12 个字节（如果 int 的长度为 4 个字节），因此给这个指针加 1，指针就前进 12 个字节（参照图 3-6）。

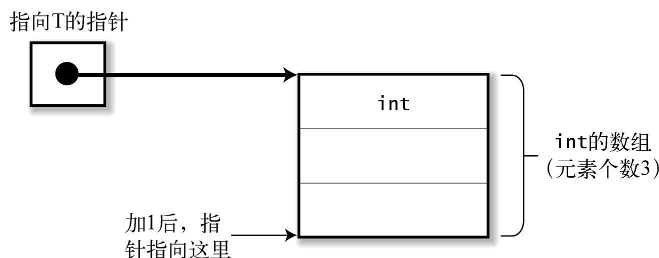


图 3-6　对“指向数组的指针”进行加法运算

道理我明白了，但是一般没有人这么用吧？

可能有人存在以上的想法。但真的有很多人就是这么用的，只不过是自己没有意识到。为什么这么说呢？在后面的章节中将会说明。

3.2.5　C 语言中不存在多维数组！

在 C 中，可以通过下面的方式声明一个多维数组：

```
int hoge[3][2];
```

我想**企图**这么干的人应该很多。请大家回忆一下 C 的声明的解读方法，上面的声明应该怎样解读呢？

是“int 类型的多维数组”吗？

这是不对的。应该是“int 的数组（元素个数 2）的数组（元素个数 3）”。

也就是说，即使 C 中存在“数组的数组”，也**不存在多维数组**[*]。

* 在 C 标准中，“多维数组”这个词最初出现在脚注中，之后这个词也会不时地出现在各个角落。尽管“不存在多维数组”这个观点会让人感觉有些极端，但如果你不接受这个观点，对于 C 的类型模型的理解，可能就会比较困难。

“数组”就是将一定个数的类型进行排列而得到的类型。“数组的数组”也只不过是派生源的类型恰好为数组。图 3-7 是“int 的数组（元素个数 2）的数组（元素个数 3）”。

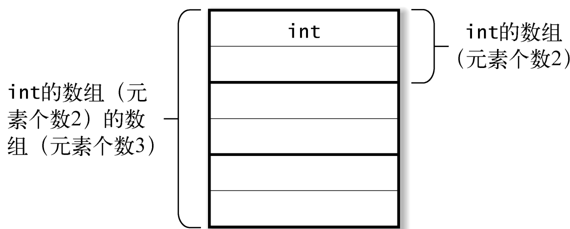


图 3-7　数组的数组

> 要　点
> **C 语言中不存在多维数组。**
> **看上去像多维数组，其实是“数组的数组”。**

对于下面的这个声明：

```
int hoge[3][2];
```

可以通过 hoge[i][j]的方式去访问，此时，hoge[i]是指“int 的数组（元素个数 2）的数组（元素个数 3）”中的第 i 个元素，其类型为“int 数组（元素个数 2）”。当然，因为是在表达式中，所以在此时此刻，hoge[i]也可以被解读成“指向 int 的指针”。

关于这一点，3.3.5 节中会有更详细的说明。

那么，如果将这个“伪多维数组”作为函数的参数进行传递，会发生什么呢？

试图将“int 的数组”作为参数传递给函数，其实可以直接传递“指向 int 的指针”。这是因为**在表达式中，数组可以解释成指针**。

因此，在将“int 的数组”作为参数传递的时候，对应的函数的原型如下：

```
void func(int *hoge);
```

在“int 的数组（元素个数 2）的数组（元素个数 3）”的情况下，假设使用同样的方式来考虑，

int 的数组（元素个数 2）<u>的数组（元素个数 3）</u>

其中下划线部分，在表达式中可以解释成指针，所以可以向函数传递

<u>指向</u> int 的数组（元素个数 2）<u>的指针</u>

这样的参数，说白了它就是“指向数组的指针”。

也就是说，接收这个参数的函数的原型为：

```
void func(int (*hoge)[2]);
```

直到现在，有很多人将这个函数原型写成下面这样：

```
void func(int hoge[3][2]);
```

或者这样：

```
void func(int hoge[][2]);
```

其实，

```
void func(int (*hoge)[2]);
```

就是以上两种写法的语法糖，它和上面两种写法完全相同。

关于将数组作为参数进行传递这种的情况下的语法糖，在 3.5.1 节中会再一次进行说明。

3.2.6　函数类型派生

函数类型也是一种派生类型，“参数（类型）”是它的属性。

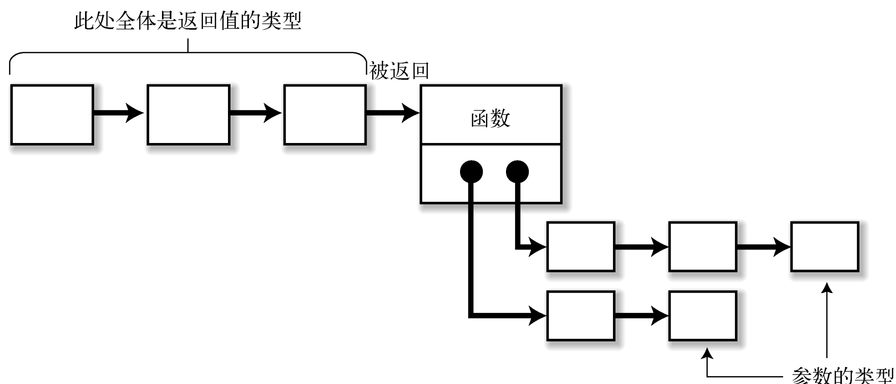


图 3-8　函数类型派生

可是，函数类型和其他派生类型有不太相同的一面。

无论是 `int` 还是 `double`，亦或数组、指针、结构体，只要是函数以外的类型，大体都可以作为变量被定义。而且，这些变量在内存占用一定的空间。因此，通过 `sizeof` 运算符可以取得它们的大小。

像这样，有特定长度的类型，在标准中称为**对象类型**。

可是，函数类型不是对象类型。因为函数没有特定长度。 所以 C 中不存在“函数类型的变量”（其实也没有必要存在）。

数组类型就是将几个派生类型排列而成的类型。因此，数组类型的全体长度为：

派生源的类型的大小×数组的元素个数

可是，函数类型是无法得到特定长度的，所以从函数类型派生出数组类型是不可能的。也就是说，不可能出现“函数的数组”这样的类型。

可以有“指向函数的指针”类型，但不幸的是，对指向函数类型的指针不能做指针运算，因为我们无法得到当前指针类型的大小。

此外，函数类型也不能成为结构体和共用体的成员。

总而言之：

从函数类型是不能派生出除了指针类型之外的其他任何类型的。

不过“指向函数的指针类型”，可以组合成指针或者作为结构体、共用体的

成员。毕竟“指向函数的指针类型”也是指针类型，而指针类型又是对象类型。

另外，函数类型也不可以从数组类型派生。

可以通过“返回～的函数”的方式派生出函数类型，不过在 C 中，数组是不能作为函数返回值返回的（参照 1.1.8 节）。

> 要　点
> **从函数类型是不能派生出除了指针类型之外的其他任何类型的。**
> **从数组类型是不能派生出函数类型的。**

3.2.7　计算类型的大小

除了函数类型和不完全类型（参照 3.2.10 节），其他类型都有大小。

通过

```
sizeof(类型名)
```

编译器可以为我们计算当前类型的大小，无论是多么复杂的类型。

```
printf("size..%d\n", sizeof(int(*[5])(double)));
```

以上的语句表示输出

```
指向返回 int 的函数（参数为 double）的指针的数组(元素个数 5)的大小。
```

在这里顺便对以前的内容也做一下复习：模仿编译器的处理方式，尝试计算各种类型的大小。

另外，我们考虑使用如下构成的机器来作为处理环境。

```
int       4 个字节
double    8 个字节
指针      4 个字节
```

【注意！】

在这里我们为了说明方便，特别地对处理环境做了假定。但是，C 语言的标准并没有对 int、double 和指针的大小进行任何规定，数据类型的大小完全取决于各处理环境的具体实现。

因此，我们通常不需要去留意数据类型的物理大小，更不应该依赖数据类型大小进行编程。

可以像下面这样，以日语[①]词组的顺序计算类型的大小：

❶ 基本类型

基本类型必定依赖处理环境进行计算。

❷ 指针

指针的大小是依赖处理环境决定的。大部分情况下，指针的大小和派生源的大小没有关系，它的大小是固定的。

❸ 数组

数组的大小可以通过派生源类型的大小乘以元素个数得到。

❹ 函数

函数的大小无法计算。

现在，可以尝试计算刚才的示例。

指向返回 int 的函数（参数为 double）的指针的数组（元素个数 5）的大小。

- ❶ 指向返回 **int** 的函数（参数为 double）的指针的数组（元素个数 5）。
  因为是 int 类型，所以在当前假定的处理环境中，计算结果为 4 个字节。
- ❷ 指向返回 int **的函数（参数为 double）**的指针的数组（元素个数 5）。
  因为是函数，所以无法计算大小。
- ❸ 指向返回 int 的函数（参数为 double）**的指针**的数组（元素个数 5）。
  因为是指针，所以在当前假定的处理环境中，计算结果为 4 个字节。
- ❹ 指向返回 int 的函数（参数为 double）的指针**的数组（元素个数 5）**。
  因为是派生源的大小为 4 的“元素个数为 5 的数组”，所以计算结果为 4×5=20 个字节。

同样地，表 3-3 中整理了对各种类型的大小进行计算的结果。

① 这里同样可以用中文词组的顺序来计算。——译者注

表3-3　计算各种类型的大小

声　明	中文的表现	大　小
`int hoge;`	hoge是int	4个字节
`int hoge[10]`	hoge是int的数组	4 × 10=40个字节
`int *hoge[10];`	hoge是int的指针的数组（元素个数10）	4 × 10=40个字节
`double *hoge[10]`	hoge是double的指针的数组（元素个数10）	4 × 10=40个字节
`int hoge[2][3];`	hoge是int的数组（元素个数3）的数组（元素个数2）	4 × 3 × 2=24个字节

3.2.8　基本类型

派生类型的底层是**基本类型**。

基本类型指，char 和 int 这样的**整型**以及 float 和 double 这样的**浮点型**。这些类型加上枚举类型，统称为**算术型***。

* 在标准中，枚举类型没有包括在基本类型（basic type）中（6.1.2.5）。在 *K&R* 中，它们被混在一起了。

此外，在 C 中，通过 short int 声明一个变量，和单纯地通过 short 声明一个变量，意义是完全一样的。

对于整型和浮点型，怎样的写法是允许的，哪种写法和哪种写法的意义是相同的，这些内容非常琐碎，特整理如下（表 3-4）。

表3-4　整数型、浮点型的种类

推　荐	同义的表现
char	
signed char	
unsigned char	
short	signed short，short int，signed short int
unsigned short	unsigned short int
int	signed，signed int，无指定类型
unsigned int	unsigned
long	signed long，long int，signed long int
unsigned long	unsigned long int
float	
double	
long double	

char 和 signed char 或者 unsigned char 同义。至于默认情况下，char 究竟是有符号的还是无符号的，C 标准并没有定义，而是取决于处理环境。

根据处理环境不同，long long 等写法有可能也是允许的，这些都不在标准的约束范围之内。ANSI C 以前的 C，存在 long float 这样的写法，它和 double 同义，但是在 ANSI C 之后这种写法就被废弃了。

另外，对于这些类型的大小，sizeof(char)（包含 signed、unsigned）必定返回 1。其他的类型**全部依赖处理环境的定义**。即使是 char，在 sizeof 肯定会返回 1 的情况下，也**没有规定肯定是 8 位**，现实中也存在 char 为 9 的处理环境。

偶尔有一些不靠谱的 C 语言入门书籍会跟大家乱嚼舌头："int 的大小是依赖于硬件的，所以尽量不要使用 int。"这种观点完全是错误的。不只是 int，无论是 short 还是 long，它们的大小都依赖于处理环境。具有讽刺意味的是，几乎所有持有这种观点的入门书的例程也在使用 int。言行不一？说话不算数？哎，我都不知道该怎么说了……

不过标准还是规定了每种类型可以表示的值的范围：

- 有符号 char，±127
- 无符号 char，0～255
- 有符号 int，有符号 short，±32767
- 无符号 int，unsigned short，0～65535
- 有符号 long，±2147483647
- 无符号 long，0～4294967295

请注意在这里有符号 int 的最小值**不是**–32768，这是因为考虑到有的机器对于负数使用 1 的补码[①]。

3.2.9　结构体和共用体

在语法上，结构体和共用体是作为派生类型使用的。

可是直到现在，我们还没有专门去说明结构体和共用体。其理由如下：

- 虽然结构体和共用体在语法上属于派生类型，但是在声明中它和数据类型修饰符（也就是 int、double 等）处于相同的位置。
- 只有派生指针、数组和函数的时候，类型才可以通过一维链表表示。结构体、共用体派生类型只能用树结构进行表现。

---

① 对于有符号的 int 值，原码的范围为–32767/1111111111111111～+32767/0111111111111111，补码的范围为–32768/1000000000000000～+32767/0111111111111111。——译者注

结构体类型可以集合几个其他不同的类型，而数组只能线性地包含同一个类型。

共用体的语法和结构体相似，但是，结构体的成员是“排列地”分配在内存中，而共用体的成员则是“重叠地”分配在内存中。在第 5 章将会介绍共用体的用途。

让我们通过图 3-9，尝试使用“类型链的方式”来表现结构体和共用体的派生。

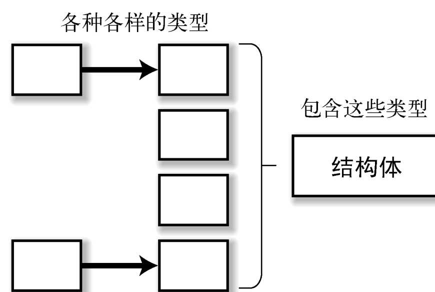


图 3-9　结构体类型的派生

3.2.10　不完全类型

不完全类型指“函数之外、类型的大小不能被确定的类型”。

总结一下，C 的类型分为：

- 对象类型（char、int、数组、指针、结构体等）
- 函数类型
- 不完全类型

结构体标记的声明就是一个不完全类型的典型例子。

对于男性（Man），他可能有妻子（wife）。如果是未婚男性，wife 就是 NULL。所以，Man 这样的类型，可以声明成下面这样：

```
struct Man_tag {
        ⋮
    struct Woman_tag *wife;  /*妻*/
        ⋮
};
```

作为妻子，可以这样声明：

```
struct Woman_tag {
        ⋮
    struct Man_tag *husband; /*夫*/
        ⋮
};
```

这种情况下，struct Man_tag 和 struct Woman_tag 是相互引用的，所以无论先声明哪一边都很麻烦。

可以像下面这样通过先声明结构体标记来回避以上问题：

```
struct Woman_tag;     ←— 将 tag 提前声明

struct Man_tag {
        ⋮
    struct Woman_tag *wife;  /* 妻 */
        ⋮
};
struct Woman_tag {
        ⋮
    struct Man_tag *husband; /* 夫 */
        ⋮
};
```

在我的环境中，结构体必须使用 typedef，所以，

```
typedef struct Woman_tag Woman;    ←— 提前对 tag 进行类型定义

typedef struct {
        ⋮
    Woman *wife;  /* 妻 */
        ⋮
} Man;

struct Woman_tag {
        ⋮
    Man *husband; /* 夫 */
        ⋮
};
```

对这种情况，在 Woman 类型的标记被声明的时候，还不知道其内容，所以无法确定它的大小。这样的类型就称为**不完全类型**。

因为不能确定大小，所以不能将不完全类型变成数组，也不能将其作为结构体的成员，或者声明为变量。但如果仅仅是用于取得指针，是可以使用不完全类型的。上面的结构体 Man，就是将 Woman 类型的指针作为它的成员。

之后，在定义 struct Woman_tag 的内容的时候，Woman 就不是不完全类型了。

在 C 标准中，void 类型也被归类为不完全类型。

3.3　表达式

3.3.1　表达式和数据类型

直到现在，我们没有进行明确地定义就使用了**表达式**（expression）这个词。

首先介绍**基本表达式**（primary expression），基本表达式是指：

- 标识符（变量名、函数名）
- 常量（包括整数常量和浮点数常量）
- 字符串常量（使用“”括起来的字符串）
- 使用()括起来的表示式

此外，对表达式使用运算符，或通过运算符将表达式和表达式相互连接，这些表示方法也称为表达式。

也就是说，“5”、“hoge”都是表达式（如果已经声明了以 hoge 作为名称的变量）。此外，“5 + hoge”也是表达式。

对于下面的表达式，

```
a + b * 3 / (4 + c)
```

它可以表现成如图 3-10 这样的树结构，这个树结构的所有部分的树*都是表达式。

* 这里指某个特定节点以下的树。

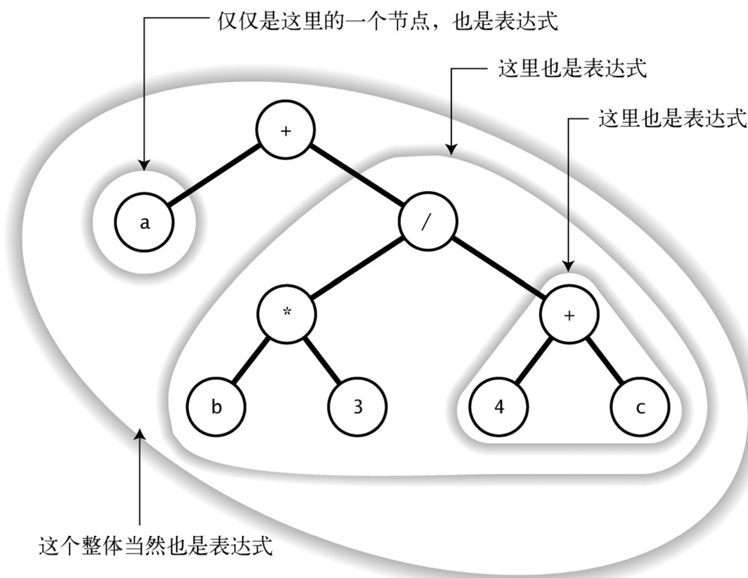


图 3-10　表达式的树结构

此外，**所有的表达式都持有类型**。

3.2 节中介绍了可以通过链的结构表现类型。如果所有的表达式都持有类型，那么对于表现表达式的树结构的节点，都可以被挂接上表现类型的链（参照图 3-11）。

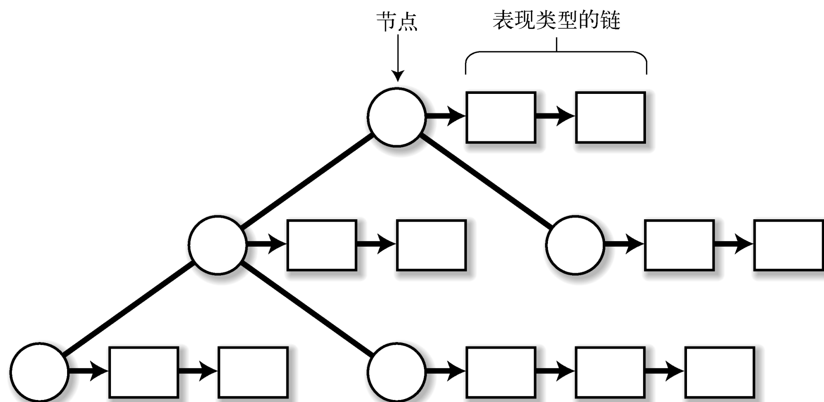


图 3-11　所有的表达式都持有类型

在对表达式使用运算符，或者将表达式作为参数传递给函数的时候，表达式中持有的类型具有特别重要的意义。

比如，对于下面这样的数组：

```
char str[256];
```

在输出这个字符数组的内容的时候，使用

```
printf(str);
```

初学者看到这段程序，难免会想：“printf()能这么写吗？”

确实，就像这个世上最有名的程序中写的这样：

```
printf("hello, world\n");
```

第 1 个参数总是传递字符串常量。

可是，在 stdio.h 的原型声明中，printf()的第 1 参数被定义为“指向 char 指针”。

字符串常量的类型为“char 的数组”，因为是在表达式中，所以它也可以当成“指向 char 的指针”。因此，字符串常量可以传递给 printf()。同样地，str 是“char 的数组”，因为是在表达式中，所以也可以当成“指向 char 的指针”，能够传递给 printf()也是很自然的事。

如果只是单纯地输出字符串，对于字符串中包含%感到麻烦，与其使用

```
printf("%s", str);
```

不如使用 puts()会更好。这个就扯远了。

此外，下面的写法也许会让某些人感到惊奇：

```
"01234567890ABCDEF"[index]
```

但如果写成这样：

```
str[index]
```

谁都不会觉得奇怪了吧。在表达式中，str 和字符串常量都属于“指向 char 的指针”，它们都可以作为[]运算符的操作数*。

* 运算符（operator）的作用对象被称为**操作数**。比如，1 + 2 这个表达式，1 和 2 是运算符 + 的操作数。



补充　针对“表达式”使用 sizeof

sizeof 运算符有两种使用方法。

一种是：

```
sizeof(类型名)
```

另外一种是：

```
sizeof 表达式
```

后者能够返回对象表达式的类型的大小。

在实际开发中，“sizeof 表达式”这种使用方式的唯一用途，就是从编译器获取数组的长度。

对于下面的声明，

```
int hoge[10];
```

在 sizeof(int)为 4 的处理环境中，

```
sizeof(hoge)
```

返回 40。因此将这个结果除以 sizeof(int)就可以得到数组元素的个数。

如果像这个例子这样显式地指定了数组的大小，即使不使用 sizeof，使用#define 给大小定义一个合适的名称也是可以满足需求的。不过在下面的情况下，使用 sizeof 也许更加方便。

```
char *color_name[] = {
    "black",
    "blue",
      :
      :
};

#define COLOR_NUM (sizeof(color_name) / sizeof(char*))
```

在这种情况下，由于使用了数组初始化表达式，这里可以省略定义数组元素的个数*，因此就不需要使用#define 定义一个固定的常量了。另外，在某些情况下需要在 color_name 中追加更多的元素，如果使用 sizeof，只需修改程序的一个地方。

* 参照 3.5.2 节。

无论怎样，sizeof 运算符只是向编译器问询大小的信息，所以，**它只能在编译器明确知道对象大小的情况下使用**。

```
extern int hoge[];
```

以上的情况是不能使用 sizeof 的。此外，

```
void func(int hoge[])
{
    printf("%d\n", sizeof(hoge));
}
```

这样的程序，也只是输出指针的长度（参照 3.5.1 节）。

也许很多人并不知道，对于“sizeof 表达式”，其实不需要括号。当然，加上括号也可以，此时的括号只是单纯地起到括起操作数的作用。

有人尽管知道这一点，但为了阅读的方便，依然使用了括号。

3.3.2 “左值”是什么——变量的两张面孔

假设有下面这样一个声明：

```
int hoge;
```

因为此时 hoge 是 int 类型，所以，只要是可以写 int 类型的值的地方，hoge 就可以像常量一样使用。

比如，将 5 赋予 hoge 之后，下面的语句

```
piyo = hoge * 10;
```

理所当然地可以写成

```
piyo = 5 * 10;
```

但是，在

```
hoge = 10;
```

的情况下，即使此时 hoge 的值为 5，

```
5 = 10;
```

这样的置换也是非法的。

也就是说，作为变量，它有作为“自身的值”使用和作为“自身的内存区域”使用两种情况。

此外在 C 中，即使不是变量名，表达式也可以代表“某个变量的内存区域”。比如这种情况：

```
hoge_p = &hoge;
*hoge_p = 10;     ←—— *hoge_p 是指 hoge 的内存区域
```

像这样，表达式代表某处的内存区域的时候，我们称当前的表达式为**左值**（lvalue）；相对的是，表达式只是代表值的时候，我们称当前的表达式为**右值**。

表达式中有时候存在左值，有时候不存在左值。比如，根据上下文，表达式可以作为左值或者右值使用，但是 5 这样的常量，或者 `1 + hoge` 这样的表达式却只能解释成右值。



补充　“左值”这个词汇的由来

在 C 以前的语言中，因为表达式在赋值的左边，所以表达式被解释成左值。“左”在英语中是 left，left value 就被简写成 lvalue。

但在 C 中，`++hoge` 这样写法也是合法的，此时 `hoge` 是指某处的内存区域，但是怎么看也看不出“左边”的意思。因此，左值这个词真有点让人摸不着头脑。

在标准委员会的定义中，lvalue 的 l 不是 left 的意思，而表示 locator（指示位置的事物）。Rationale 中有下面一段描述，

The Committee has adopted the definition of lvalue as an object locator.

尽管如此，JIS X3010 还是将 lvalue 解释成了“左值”。[①]

3.3.3　将数组解读成指针

正如在前面翻来覆去提到的那样，在表达式中，数组可以解读成指针。

```
int hoge[10];
```

以上的声明中，`hoge` 等同于`&hoge[0]`。

`hoge` 原本的类型为“`int` 的数组（元素个数 `10`）”，但并不妨碍将其类型分类“数组”变换为“指针”。

图 3-12 表现了其变换的过程。

① 中国国家标准 GB/T 15272–94（189 页）中，也是将 lvalue 解释成左值。——译者注

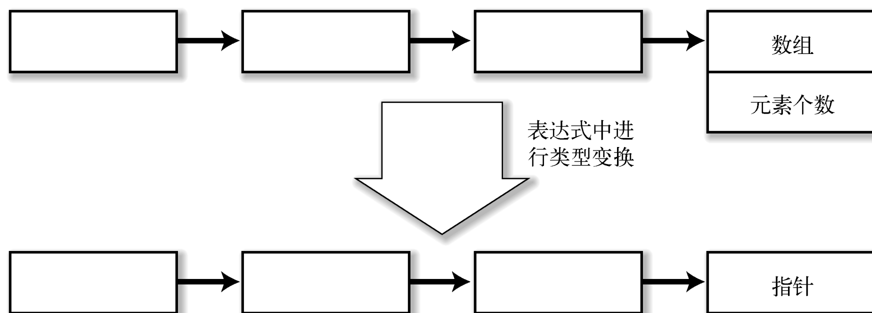


图 3-12　将数组解读成指针

此外，数组被解读成指针的时候，该指针不能作为左值。

这个规则有以下的例外情况。

❶ 数组为 sizeof 运算符的操作数

在通过“sizeof 表达式”的方式使用 sizeof 运算符的情况下，如果操作数是“表达式”，此时即使对数组使用 sizeof，数组也会被当成指针，得到的结果也只是指针自身的长度。照理来分析，应该是这样的吧？可是，当数组成为 sizeof 的操作数时，“数组解读为指针”这个规则会被抑制，此时返回的是数组全体的大小。请参照 3.3.1 节的补充内容。

❷ 数组为&运算符的操作数

通过对数组使用&，可以返回指向整体数组的指针。在 3.2.4 节中已经介绍了“指向数组的指针”。

这个规则已经被追加到 ANSI C 规则之中。此前的编译器，在对数组使用&的时候，大多会报错。因此，当时的程序在这一点上不会出现问题。那么这个规则的制定有什么好处呢？我想应该是为了保持统一吧。

❸ 初始化数组时的字符串常量

我们都知道字符串常量是“char 的数组”，在表达式中它通常被解读成“指向 char 的指针”。其实，初始化 char 的**数组**时的字符串常量，作为在花括号中将字符用逗号分开的初始化表达式的省略形式，会被编译器特别解释（参照 3.5.3 节）。

在初始化 char 的指针的时候，字符串常量的特别之处，需要引起注意。

3.3.4　数组和指针相关的运算符

以下介绍数组和指针相关的运算符。

▲解引用

单目运算符*被称为**解引用**。

运算符*将指针作为操作数，返回指针所指向的对象或者函数。只要不是返回函数，运算符*的结果都是左值。

从运算符*的操作数的类型中仅仅去掉一个指针后的类型，就是运算符*返回的表达式的类型（参照图3-13）。

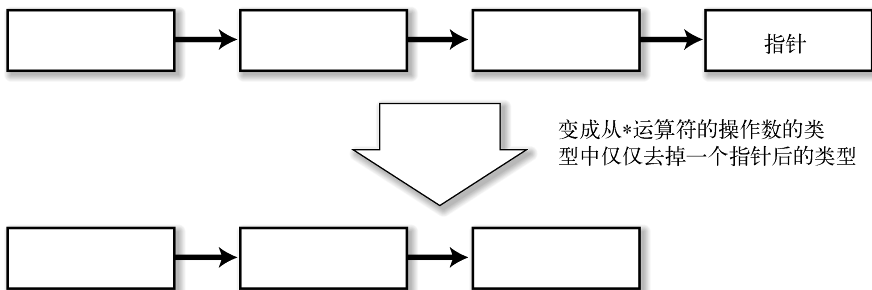


图3-13　使用解引用而发生的类型的变化

▲地址运算符

单目运算符&被称为**地址运算符**。

&将一个左值作为操作数，返回指向该左值的指针。对左值的类型加上一个指针，就是&运算符的返回类型（参照图3-14）。

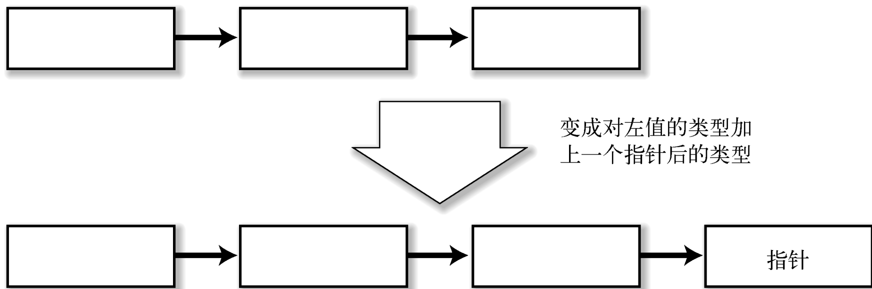


图3-14　使用地址运算符而发生的类型的变化

地址运算符不能将非左值的表达式作为操作数。

▲下标运算符

后置运算符[]被称为**下标运算符**。

[]将**指针**和整数作为操作数。

```
p[i]
```

是

```
*(p + i)
```

的语法糖，除此以外没有任何其他意义。

对于声明为 int a[10]的数组，使用 a[i]的方式进行访问的时候，由于 a 在表达式中，因此它可以被解读成指针。所以，你可以通过下标运算符访问数组（将指针和整数作为操作数）。

归根结底，p[i]这个表达式就是*(p + i)，所以下标运算符返回的类型是，从 p 的类型去掉一个指针的类型。

▲->运算符

在标准中，似乎并没有定义->运算符的名称，现实中有时它被称为“箭头运算符”。

通过指针访问结构体的成员的时候，会使用->运算符。

```
p->hoge;
```

是

```
(*p).hoge;
```

的语法糖。

利用*p 的*，从指针 p 获得结构体的实体，然后引用成员 hoge。

3.3.5　多维数组

在 3.2.5 节中，我们提到了 **C 语言中不存在多维数组**。

那些看起来像多维数组的其实是“数组的数组”。

这个“多维数组”（山寨货），通常使用 hoge[i][j]的方式进行访问。让我们来看一看这个过程中究竟发生了什么。

```
int hoge[3][5];
```

对于上面这个“数组的数组”，使用 hoge[i][j]这样的方式进行访问（参照图 3-15）。

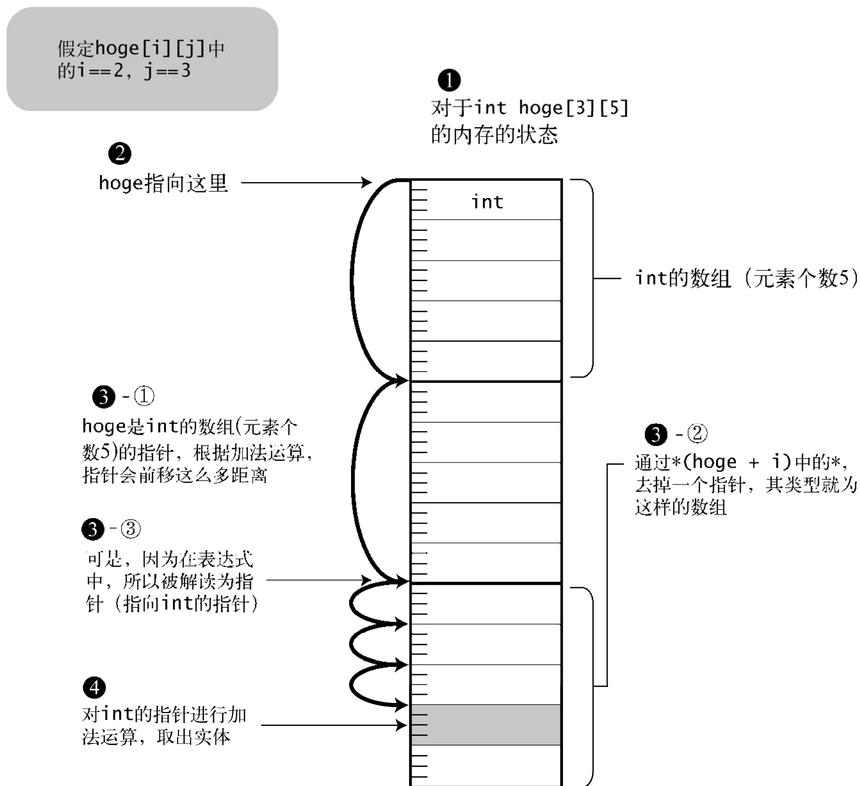


图 3-15　访问多维数组

❶ hoge 的类型为“int 的数组（元素个数 5）的数组（元素个数 3）”。

❷ 尽管如此，**在表达式中数组可以被解读成指针**。因此，hoge 的类型为“指向 int 的数组（元素个数 5）的指针”。

❸ hoge[i]是*(hoge + i)的语法糖。

① 给指针加上 i，就意味着指针前移它指向的类型×i 的距离。hoge 指向的类型为“int 的数组（元素个数 5）”，因此，hoge + i 让指针前移了 sizeof(int[5])×i 的距离。

② 通过*(hoge + i)中的*，去掉一个指针，*(hoge + i)的类型就是“指向 int 的数组（元素个数 5）”。

③ 尽管如此，由于在表达式中，数组可以解读成指针，所以*(hoge + i)的最终类型为“指向 int 的指针”。

❹ (*(hoge + i))[j]和*((*(hoge + i)) + j)其实是相等的，因此，(*(hoge + i))[j]就是“对指向 int 的指针加上 j 后得到的地址上的内容”，其类型为 int。

某些语言中，使用 array[i, j]这样的写法来支持多维数组*。

* Pascal 就支持这样的写法，但是，Pascal 的“多维数组”只不过是“数组的数组”的语法糖。

在 C 中，因为没有多维数组，所以使用“数组的数组”来代替，这样倒也没有什么问题。可是，如果反过来，只有多维数组，而没有“数组的数组”，事情就麻烦了。

比如，将某人一年之中每天的工作时间使用下面这个“数组的数组”来表现*，

* 数组中，月和日是从 0 开始的，只有在输出的时候才可修正。2 月等情况下，数组的元素就会显得冗余，但是不会产生问题。

```
int working_time[12][31];
```

在这里，如果开发一个根据一个月的工作时间计算工资的函数，可以像下面这样将某月的工作时间传递给这个函数，

```
calc_salary(working_time[month]);
```

calc_salary 的原型像下面这样：

```
int calc_salary(int *working_time);
```

这种技巧只有通过“数组的数组”才能实现，多维数组是无能为力的。



补充　运算符的优先级

C 语言中有数量众多的运算符，其优先级分 15 个级别。

和其他语言相比，C 语言的优先级别还是非常多的。很多 C 的参考书，都像表 3-5 这样记载了运算符优先级的内容。

其中，对于优先级“最高”的是()，有相当多的人抱有以下观点：

> 当需要改变原本语法规定的优先级、强制地设定自己需要的优先级的时候，程序员们会使用()。因此，()具有最高的优先级是理所当然的。

这是一种**误解**。

如果()可以这样理解，还有必要特地将它记载在优先级的表中吗？

这个表中的()，（正如 *K&R* 中记述的那样）代表着调用函数的运算符。此时的优先级是指，对于 func(a, b)这样的表达式，func 和()之间的关联强度。

表3-5　运算符优先顺序表（摘录于*K&R* p.65）

运　算　符	连接规则
() [] -> .	从左往右
! ~ ++ -- + - * & (*type*) sizeof	从右往左
* / %	从左往右
+ -	从左往右
<< >>	从左往右
< <= > >=	从左往右
== !=	从左往右
&	从左往右
^	从左往右
¦	从左往右
&&	从左往右
¦¦	从左往右
?:	从右往左
= += -= *= /= %= &= ^= ¦= <<= >>=	从右往左
,	从左往右

注：进行单目运算的+、-、&和*的优先级高于进行双目运算的+、-、&和*。

此外，我们经常看到类似于下面这样的编码：

```
*p++;
```

究竟是对 p 进行加法运算？还是对 p 所指向的对象（*p）进行加法运算？关于这一点，很多书是这样解释的：

> 尽管*和++的优先级相同，但由于连接规则是从右往左，所以 p 和++先进行连接。因此，被进行加法运算的不是*p，而是 p。

就连 *K&R* 自身也是这样说明的。其实这种说法不太恰当。

根据 BNF（Backus-Naur Form）规则，C 语言标准定义了语法规则，其中也包含了运算符的语法规则。

关于 BNF，由于超出了本书的范围，在此就不做说明了。根据 BNF，后置的++比前置的++和*等运算符的优先级高，()和[]的优先级相同*。

* 参照 *K&R* 中的标准 6.3.2 和 6.3.3。

也就是说，关于*p++的运算符优先级，

> 后置的++比*的优先级高，因此，被进行加法运算的不是*p，而是 p。

从语法上来看，这种说法才是比较合理的。

3.4 解读 C 的声明（续）

3.4.1 const 修饰符

const 是在 ANSI C 中追加的修饰符，它将类型修饰为“只读”。

名不副实的是，const **不一定代表常量**。const 主要被用于修饰函数的参数。将一个常量传递给函数是没有意义的。无论怎样，使用 const 修饰符（变量名），只意味着使其“只读”。

```
/* const 参数的范例 */
char *strcpy(char *dest, const char *src);
```

strcpy 是持有被 const 修饰的参数的范例。此时，所谓的“只读”是如何表现的呢？

做个实验应该很快就会明白，上面例子中的 src 这个变量**没有定义为只读**。

```
char *my_strcpy(char *dest, const char *src)
{
    src = NULL;    ←即使对 src 赋值，编译器也没有报错
}
```

此时，成为只读的不是 src，而是 src 所指向的对象。

```
char *my_strcpy(char *dest, const char *src)
{
    *src = 'a';    ←ERROR!!
}
```

如果将 src 自身定义为只读，需要写成下面这样：

```
char *my_strcpy(char *dest, char * const src)
{
    src = NULL;    ←ERROR!!
}
```

如果将 src 和 src 指向的对象都定义为只读，可以写成下面这样：

```
char *my_strcpy(char *dest, const char * const src)
{
    src = NULL;    ←ERROR!!
    *src = 'a';    ←ERROR!!
}
```

在现实中，当指针作为参数时，const 常用于将指针指向的对象设定为只读。

通常，C 的参数都是传值。因此，无论被调用方对参数进行怎样的修改，都不会对调用方造成任何影响。如果想要影响调用方的变量（通过函数参数将函数内的一些值返回），可以将指针作为参数传递给函数。

可是，在上面的例子（my_strcpy）中，传递的是 src 这个指针。其本来的意图是想要传递字符串（也就是 char 的数组）的值，由于在 C 中数组是不能作为参数传递的，情非得已才不得不将指向初始元素的指针传递给函数（因为数组可能会很大，所以传递指针有益于提高程序的效率）。

产生的问题是，为了达到从函数返回值的目的，需要向函数传递一个指针，这种方式让人感觉有些混乱。

此时，考虑在原型声明中加入 const，

> 尽管函数接受了作为参数的指针，但是指针指向的对象不会被修改。

也就是说：

> 函数虽然接受了指针，但是并不意味着要向调用方返回值。

strcpy()的意图就是——src 是它的输入参数，但是不允许修改它所指向的对象。

可以通过以下的规则解读 const 声明：

❶ 遵从 3.1.2 节中提到的规则，从标识符开始，使用英语由内向外顺序地解释下去。
❷ 一旦解释完毕的部分的左侧出现了 const，就在当前位置追加 read-only。
❸ 如果解释完毕的部分的左侧出现了数据类型修饰符，并且其左侧存在 const，姑且先去掉数据类型修饰符，追加 read-only。
❹ 在翻译成中文的过程中，英语不好的同学请注意：**const 修饰的是紧跟在它后面的单词**。

因此，

```
char * const src
```

可以解释成：

src is read-only pointer to char

➔src 是指向 char 的只读的指针

```
char const *src
```

可以解释成：

src is pointer to read-only char

➔src 是指向只读的 char 的指针

此外，容易造成混乱的是，

```
char const *src
```

和

```
const char *src
```

的意思完全相同。

3.4.2 如何使用 const？可以使用到什么程度？

很多人习惯在函数注释的参数说明部分，使用(i)、(o)、(i/o)等标记[1]。

这里举一个有些矫揉造作的例子。

```
/*****************************************************
 *  void search_point(char *name, double *x, double *y)
 *
 *  功能：将名称作为key，检索“点”，返回坐标
 *  参数：(i) name 名称（检索key）
 *        (o) x x坐标
 *        (o) y y坐标
 *****************************************************
```

唉，每次要对函数做这样的注释，是不是有点麻烦？于是有很多人会将别的函数的注释一成不变地复制过来，事后还鬼话连篇说自己忘了修改了。其实这些人就没把注释当回事儿，他们认为直接看代码就什么都可以明白，老是揪着注释的问题不放，简直就是没事找事。

在这里的注释中，虽然标记了各参数是(i)还是(o)，但编译器可不会注意到这些。

对此，search_point 原型使用下面的方式进行声明：

```
void search_point(char const *name, double *x, double *y);
```

当你错误地向 name[i]赋值的时候，编译器会很负责任地向我们提出警告。

* 如果不用 const，而是使用(i)并且很放心地将指针就这样传递给了函数，之后即使变量被改写了，你也很难马上发现。

① （i）指用于输入的参数，（o）指用于输出的参数，（i/o）指用于输入输出的参数。——译者注

因此，比起在注释中标记什么(i)或者(o)，使用 const 可靠性会提高很多*。

对于上面的 char const *name，是不能将它赋予 char*类型的变量的（除非强制转型）。其中的理由显而易见：如果单纯地将它赋予 char*类型的变量，之后就可以改写 name 所指向的对象的内容，const 的意义就丧失殆尽了。

同理，将 char const *类型的指针传递给使用 char*作为参数的函数，也是不允许的。因此，一旦给指针类型的参数设定了 const，当前层次以下的函数就必须全部使用 const*。

* 在一些通用函数中，如果本应该是 const 的参数却没有加上 const，会经常导致调用方无法使用 const。

假设有这样一个结构体：

```
typedef struct {
    char  *title;     /*标题*/
    int   price;      /*价格*/
    char  isbn[32];   /*ISBN*/
          ⋮
} BookData;
```

将上面这个结构体作为输入参数的函数原型，可以写成下面这样：

```
/*注册书的数据*/
void regist_book(BookData const  book_data);
```

因为使用了 const，所以 book_data 所指向的对象是禁止改写的。好吧，现在可以放心地将 BookData 传递给这个函数了……

不幸的是，我们发现被传递的数据中，书的标题（book_data->title）所指向的内容**是可以被改写的**。

之所以发生这样的事情，是因为根据指定的 const 而成为“只读”的对象只是“book_data 所指向的对象自身”，而不包括“book_data 所指向的对象再向前追溯到的对象”（参照图 3-16）。

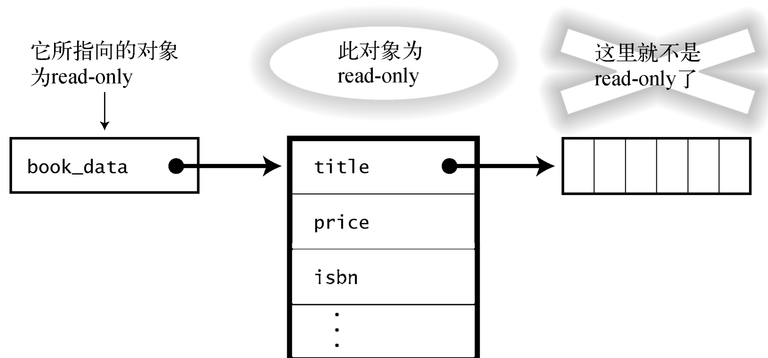


图 3-16　const 的边界

因此，如果将结构体 BookData 修改成这样：

```
typedef struct {
                 ↙试着将*title 设定为 const
    char const *title;     /*标题*/
    int   price;      /*价格*/
    char  isbn[32];   /*ISBN*/
      ⋮
} BookData;
```

这一次，**就算是上帝来了**也改不了 title 所指向的对象了*。

正因为如此，很多人对 const 究竟能为现实中的编程提供多少便利持怀疑态度。

* Java 的 String 类也是不可变(immutable)的。这种方式当然有它的益处，但是有时候它也会给你带来麻烦。



补充　const 可以代替#define 吗？

通常，C 语言使用预处理器的宏功能定义常量，就像下面这样：

```
#define HOGE_SIZE (100)
          ⋮
int hoge[HOGE_SIZE];
```

可是，预处理器是独立于 C 语言语法的，因此在调试的时候时常会出现一些问题。由宏定义自身的问题造成的错误往往爆发在使用它的地方，这给纠错工作带来很大的困难。

惹不起，总躲得起吧？大不了不使用“坑爹”的宏，是不是可以写成下面这样：

```
const int HOGE_SIZE = 100;

int hoge[HOGE_SIZE];
```

亲，写成这样还是不行!!

尽管在 C 中，数组的元素个数必须为常量*，但无论怎样，const 修饰的标识符不是常量，它只是“只读”而已。因此，上面的写法还是错误的*。

* 但是，ISO C99 没有这样的规定。

* C++另当别论。

3.4.3 typedef

typedef 用于给某类型定义别名。

比如，

```
typedef char *String;
```

通过以上的声明，以后对于“指向 char 的指针”可以使用“String”这个别名。

可以按照普通的变量声明的顺序来解释 typedef。对于上面的“String”，如果像对待变量名一样用英语的顺序进行解释，应该是下面这句话：

String is pointer to char
　➔String 是指向 char 的指针

由此，String 作为“指向 char 的指针”这个类型的别名被声明。

之后，在使用 String 时，你可以写成这样：

```
String hoge[10];
```

它的意思是：

hoge is array（元素个数 10）of String;
　➔ hoge 是 String 的数组（元素个数 10）

如果将 String 和被定义成 String 类型的指向 char 的指针机械地进行置换，就会产生下面的解释：

hoge is array（元素个数 10）of pointer to char;
　➔ hoge 是指向 char 的指针的数组（元素个数 10）

语法上，typedef 属于“存储类型修饰符”（参照 2.2.1 节的补充内容）。可是无论怎么看也看不出 typedef 指定了“存储类别”。其实与此无关，typedef 之所以被划分为存储类型修饰符，应该是由于指定类型的语法沿用了通常声明标识符的语法规则。

> 要　点
> **typedef 使用和通常的标识符声明相同的方式进行解释。**
> **可是，被声明的不是变量或者函数，而是类型的别名。**

平时在声明结构体的时候，我肯定会指定 typedef。顺便提一下，此时我会尽可能省略 tag*。

* 也有人鄙视这样的风格……

```
typedef struct {
    ⋮
} Hoge;
```

这个声明没有什么特别的地方，关于结构体，假设写成下面这样：

```
struct Hoge_tag {
    ⋮
} hoge;
```

由于可以声明 struct Hoge_tag 类型的变量 hoge，如果将和这个变量名对应的部分置换成类型的名称，就变成了 typedef 的声明了。

此外，在声明变量的时候，可以像下面这样一次性声明多个变量：

```
int a, b;
```

同样地，typedef 也可以一次声明类型的多个别名。

可是这么做，除了让声明难以阅读之外，你得不到任何好处。偶尔也会见到下面这样的声明：

```
typedef struct {
    ⋮
} Hoge, *HogeP;
```

这个声明其实和下面的声明效果相同：

```
typedef struct {
    ⋮
} Hoge;

typedef Hoge *HogeP;
```

3.5 其他

3.5.1 函数的形参的声明

C 语言可以像下面这样声明函数的形参：

```
void func(int a[])
{
    ⋮
}
```

对于这种写法，无论怎么看都好像要向函数的参数传递数组。

可是，在 C 中是不能够将数组作为函数的参数进行传递的。无论如何，

在这种情况下，你只能传递指向数组初始元素的指针。

在声明函数形参时，作为类型分类的数组，可以被解读成指针。

```
void func(int a[])
{
    ⋮
}
```

可以被自动地解读成

```
void func(int *a)
{
    ⋮
}
```

此时，就算你定义了数组的元素个数，也会被无视。

必须引起注意的是，在 C 语言中，**只有在这种情况下**，int a[]和 int *a 才具有相同的意义。请同时参照 3.5.2 节。

> 要　点
> **【非常重要!!】**
> **只有在声明函数形参的情况下，int a[]和 int *a 才具有相同的意义。**

下面是一个稍微复杂一点的形参声明的例子：

```
void func(int a[][5])
```

a 的类型为“int 的数组（元素个数 5）的数组（元素个数不明）”，因此它可以解读成“指向 int 数组（元素个数 5）的指针”。因此，上面的声明本来的意思是：

```
void func(int (*a)[5])
```



补充　*K&R* 中关于函数形参声明的说明

K&R 的 p.121 中，有下面这样一段记述：

作为函数定义的形参，

```
char s[];
```

和

```
char *s;
```

是完全相同的。我们认为写成后面这样比较好，因为这种写法能更加明确地表示这里的参数是一个指针。

这段文字本身可能并没有什么问题。可是在 *K&R* 中，这段文字是在说明了`*(pa + i)`和`pa[i]`之后唐突地出现的。因此，读者非常容易读漏掉前面的“作为函数定义的形参”这个前提条件*。

此外，在这个例子中，为什么**右边加上了分号***？ANSI C 中，定义形参的时候一般是不加分号的，**莫非早期的 C 语言就是这样的**？还是忘了修正第一版的内容？

更让人费解的是，在 *K&R* 中，接着还有下面这段文字，

> 在向函数传递数组名的时候，函数会根据情况判断它是作为数组传入的，还是作为指针传入的，并进行相应的操作。

至少对于我来说，真的是**完全不明白这段文字的意思**。

真相应该是：

> 对于 C 语言，在表达式中的数组可以被解读成“指向初始元素的指针”
>
> 
>
> 函数的参数也是表达式，所以，此时的数组也可以被解读成“指向初始元素的指针”
>
> 
>
> 因此，向函数传递的往往是指针。

C 也**不具备**“函数会根据情况判断它是作为数组传入的，还是作为指针传入的，并进行相应的操作”这么神的超能力。只是指针经常被作为参数向函数传递罢了。

实际上，对于刚才那段引用中的

> 我们认为写成后面这样比较好，因为这种写法能更加明确地表示这里的参数是一个指针。

* 此外，原书中“作为函数定义的形参”（As formal parameters in a function definition）**这句话正好到了页尾**，这又增大了读者漏读的可能性。

* 这应该不是翻译过程中的问题，原书中就已经加上了分号。

这段话，让人费解的是：

> 如果 C 语言的作者认为后面的写法比较好，为什么还故意加上“只有在函数形参中，数组的声明才可以被解读为指针”这么个**奇怪的规则**呢？

关于这一点，“The Development of the C Language”[5]中有这样一段说明：

> Moreover, some rules designed to ease early transitions contributed to later confusion. For example, the empty square brackets in the function declaration
>
> ```
> int f(a) int a[]; { ... }
> ```
>
> are a living fossil, a remnant of NB's way of declaring a pointer;

翻译成中文是：

> 为了容易地进行早期的移植而设计的几个规则，之后带来了一些混乱。比如函数声明*的空方括号，
>
> ```
> int f(a) int a[]; { ... }
> ```
>
> 就是一个活化石，NB（New B）的指针声明方法留下的后遗症。

* 这是 ANSI C 以前的方式。

3.5.2　关于空的下标运算符[]

在 C 语言中，遇到以下情况下标运算符[]可以将元素个数省略不写。

对于这些情况，**不同编译器会有各自特别的解释**，所以不能作为普遍的规则来使用。

❶ 函数形参的声明

正如 3.5.1 节中说明的那样，对于函数的形参，最外层的数组会被解读成指针，即使定义了元素个数也会被无视。

❷ 根据初始化表达式可以确定数组大小的情况

在下面的情况下，编译器可以根据初始化表达式来确定元素的个数，所以可以省略**最外层数组**的元素个数。

```
int    a[] = {1, 2, 3, 4, 5};
char   str[] = "abc";
```

```
double matrix[][2] = {{1, 0}, {0, 1}};
char   *color_name[] = {
    "red",
    "green",
    "blue",
};
char   color_name[][6] = {
    "red",
    "green",
    "blue",
};
```

在初始化数组的数组的时候，如果有初始化表达式，貌似即使不是最外层的数组，编译器也应该能够确定其元素个数。可是，在C语言中，允许下面这样不整齐的数组初始化，因此还是不能简单地确定最外层数组以外的元素个数。

```
int a[][3] = { /* int a[3][3]的省略形式*/
    {1, 2, 3},
    {4, 5},
    {6}
};

char str[][5] = { /* char str[3][5]的省略形式*/
    "hoge",
    "hog",
    "ho",
};
```

似乎可以考虑让编译器选择一个最大值，但C的语法并没有这么做。

如果这么做是为了排查程序员的编程失误，那为什么没有把上面“不整齐的数组”也规定为错误？对于这种现象，我至今百思不得其解（莫非只是因为疏忽？）。

顺便说一下，在初始化上面这样不整齐的数组的时候，没有对应的初始化表达式的元素会被初始化为0。

❸使用 extern 声明全局变量的情况

全局变量在多个编译单元（.c文件）中的某一个中**定义**，然后从其他代码文件通过 extern 进行声明。

在定义的时候还是需要元素个数的，但是在使用 extern 进行声明的时候，在连接的时候编译器可以确定实际的数组大小，所以可以省略最外层数组的元素个数。

正如前面说明的那样，只有在声明函数形参的时候，数组的声明才可以

被解读成指针。

像下面这样进行全局变量声明的时候，将数组和指针混在一起，除了程序不能正常运行之外，编译器通常也不会报告任何警告或者错误。这一点需要引起注意*。

* 如今的链接器，有时也会报错。

```
file_1.c……中
    int  a[100];

file_2.c……中
    extern int *a;
```



补充　定义和声明

在 C 语言中，“声明”在规定变量或者函数的实体的时候被称为“定义”。比如，像下面这样声明全局变量的行为，就是“定义”*。

* 准确地说，int a;这样的定义属于暂时定义（tentative definition）；int a = 0;这样的加上了初始化表达式的定义属于“外部定义”。

```
int a;
```

以下的 extern 的声明，意味着“使在某处声明的对象能够在当前的地方使用”，因此它不是“定义”。

```
extern int a;
```

同样地，函数的原型是“声明”，函数的“定义”是指写着函数的实际执行代码的部分。

自动变量的情况下，区别定义和声明是没有意义的，因为此时声明必然伴随着定义。

3.5.3 字符串常量

使用""包围起来的字符串被称为字符串常量。

字符串常量的类型是“char 的数组”，因此在表达式中，它可以解读为指针。

```
char *str;

str = "abc";     ←— 将「指向"abc"的初始元素的指针」赋给 str
```

可是，char 数组的初始化是个例外。此时的字符串常量，作为在花括号

中分开书写的初始化表达式的省略形式，编译器会进行特殊处理。

```
char str[] = "abc";
```

和

```
char str[] = {'a', 'b', 'c', '\0'};
```

具有相同的含义。

以前 C 语言只有标量，所以不能初始化自动变量的数组。因此，

```
char str[] = {'a', 'b', 'c', '\0'};
```

必须写成下面这样：

```
static char str[] = {'a', 'b', 'c', '\0'};
```

同样地，

```
char str[] = "abc";
```

这样的写法也是不允许的，你必须写成下面这样：

```
static char str[] = "abc";
```

可是，从 ANSI C 开始，即使是自动变量的数组，也可以被整合来进行初始化。

```
char str[] = "abc";
```

正因为如此，上面的写法是合法的。所以，

```
char str[4];

str = "abc";
```

这样的写法是非法的。

你是不是有点晕了？下面的例子不是初始化 char 的**数组**，而是初始化**指针**，所以也是合法的：

```
char *str = "abc";
```

此时的“abc”就是普通的“char 的数组”，在表达式中被解释成“指向 char 的指针”，然后被赋给 str。

只要按顺序对标识符的声明和初始化表达式中花括号的对应关系进行一步步分析，更复杂的例子也一定能够解释。

```
char *color_name[] = {
    "red",
```

```
    "green",
    "blue",
};
```

此时，标识符 color_name 的类型为“指向 char 的指针的数组”，类型分类“数组”对应初始化表达式的最外层的花括号。因此，“red”也好，“blue”也好，它们都是“指向 char 的指针”。

```
char color_name[][6] = {
    "red",
    "green",
    "blue",
};
```

这个例子中的 color_name 的类型为“char 的数组（元素个数 6）的数组”，同样地，类型分类“数组”，对应于初始化表达式的最外层的花括号，因此，无论是“red”，还是“blue”，它们都是“char 的数组（元素个数 6）”。所以，上面的声明和

```
char color_name[][6] = {
    {'r', 'e', 'd', '\0'},
    {'g', 'r', 'e', 'e', 'n', '\0'},
    {'b', 'l', 'u', 'e', '\0'},
};
```

具有相同的意思。

通常，字符串常量保存在只读的内存区域（准确地说，实际的保存方式还是要依赖处理环境的具体实现的）。但如果在初始化 char 的**数组**的时候，采取将原本在花括号中分开书写的初始化表达式的省略形式，并且不给数组自身指定 const，字符串常量就是可写的。

```
char str[] = "abc";

str[0] = 'd';    ←— 可写
```

但如果写成下面这样，就会报错：

```
char *str = "abc";

str[0] = 'd';    ←— 在大部分的处理环境中会报错
```



补充　　字符串常量就是 char 的数组

字符串常量的类型是“char 的数组”。
可是，在表达式中它可以被解释成“指向 char 的指针”。

是不是有很多同学认为字符串常量本来就是“指向 char 的指针”呢？通过以下的代码，可以证明字符串常量本质还是数组：

```
printf("size..%d\n", sizeof("abcdefghijklmnopqrstuvwxyz"));
```

3.5.4 关于指向函数的指针引起的混乱

正如 2.3.2 节中说明的那样，对于 C 语言，表达式中的函数可以被解读成“指向函数的指针”。

在信号处理、事件驱动的程序中，这种特性往往以回调函数的形式被使用。

```
/*如果发生 SIGSEGV（Segmentation falut），回调函数 segv_handler */
signal(SIGSEGV, segv_handler);
```

可是，如果基于之前说明过的 C 语言声明规则，int func()这样的声明会被解释为“返回 int 的函数”，如果函数在表达式中，只是取出 func 解释成“指向返回 int 函数的指针”，是不是感觉很怪异？如果一定要使用指向函数的指针，必须要写成&func。

对于上面信号处理的函数，写成

```
signal(SIGSEGV, &segv_handler);
```

这样，实际上也能**顺利地执行**。

相反，像

```
void (*func_p)();
```

这样，变量 func_p 声明为指向函数的指针，进行函数调用的时候，可以写成

```
func_p();
```

但是像 int func()这种声明，都是用 func()这样的方式进行调用的，从对称性的角度考虑，对于 void (*func_p)()，必须要写成

```
(*func_p)();*
```

这样也是能**毫无问题地执行**的。

* 早期的 C 语言中，好像也只能这么写……

是不是感觉 C 语言的关于指向函数的指针的语法比较混乱？

混乱产生的原因就是：“表达式中的函数可以解读成‘指向函数的指针’”这个意图不明的规则（难道就是为了和数组保持一致？）。

为了照顾到这种混乱，ANSI C 标准对语法做了以下例外的规定：

- 表达式中的函数自动转换成“指向函数的指针”。但是，当函数是地址运算符&或者 `sizeof` 运算符的操作数时，表达式中的函数不能变换成“指向函数的指针”。
- 函数调用运算符()的操作数不是“函数”，而是“函数的指针”。

如果对“指向函数的指针”使用解引用*，它暂时会成为函数，但是因为在表达式中，所以它会被瞬间地变回成“指向函数的指针”。

结论就是，即使对“指向函数的指针”使用*运算符，也是**对牛弹琴**，因为此时的运算符*发挥不了任何作用。

因此，下面的语句也是能**顺利执行**的，

```
(**********printf)("hello, world\n");     ←— 无论如何，*就是什么也没做
```

3.5.5　强制类型转换

`cast` 是将某类型强制地转换成其他类型的运算符，它写成下面这样：

(类型名称)

简单地说，强制类型转换有两种使用方式。

其一是基本类型的强制转换，比如像下面这样想要将 `int` 作为 `double` 来使用的情况：

```
int hoge, piyo;
       ⋮
printf("hoge / piyo..%f\n", (double)hoge / piyo);
```

在 C 中，无论怎样，**`int` 的除法运算的结果还是 `int`***，如果想要得到小数部分，上面除法运算符的某一边（或者是双方）的操作数必须转换成 `double`。

* 这可是一个很大的陷阱。

此时，强制类型转换将 `int` 类型的值转换成实际的 `double` 类型。编译器在大多数情况下，会生成强制转换对应的机器代码。

另外一个强制转换的方式就是指针类型的强制转换。

C 语言编译器对于指针类型，根据其指向的类型的不同，分别采取不同的对待方式。在运行时，无论是指向 `int` 的指针，还是指向 `double` 的指针，从机器语言的角度来看，它们在大多数的处理环境中都只是地址。所谓的指针

的强制类型转换，就是对指针进行强制读取转换。

比如像下面这样的指针强制类型转换：

```
double double_var;
int    *int_p;

int_p = (int*)&double_var;  ←—将指向 double 的指针转换成指向 int 的指针
```

一旦将“指向 double 指针”强制地转换为“指向 int 的指针”，就无法追踪指针原本指向什么对象了。

因此，写成*int_p，取出的数据类型为 int 类型，对 int_p 加 1，指针前移 sizeof(int)。

如果想要开发出可移植性高的程序，就应该避免对指针进行强制类型转换。规范的编程是不会草率地对指针进行强制类型转换的*。

* 以前，malloc()的返回值是必须要进行强制转型的，但到了 ANSI C 的时候就不需要这么做了。

可是也有一些例外，比如对于一个通用的 GUI 类库程序，界面上的按钮等控件可能会被关联各种类型的数据。此时，姑且先让控件关联到 void*，之后根据需要再将其强制转换到关联数据的本来的类型。现实中的指针强制类型转换的场景，大致也就是这种程度。

“不知道为什么编译器提示了警告，姑且先来一把强制转型”→“只要不再出现警告，就随它去了……”——这种恶习是绝对需要避免的*。

* 经常能看到这种事。

编译器是不会无端地给出警告的，强制类型转换只是暂时掩盖了问题。就算通过了编译，程序也很有可能不会正常运行，要不就是虽然在当前的环境中能正常运行，一拿到别的环境中就跑不起来了。

> 要 点
> **不要使用强制类型转换来掩盖编译器的警告。**

3.5.6 练习——挑战那些复杂的声明

应该是小试牛刀的时候了。

在 ANSI C 的标准库中，有一个 atexit()函数。如果使用这个函数，当程序正常结束的时候，可以回调一个指定的函数。

atexit()的原型定义如下：

```
int atexit(void (*func)(void));
```

❶ 首先着眼于标识符。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is

❷ 解释用于函数的()。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function() returning

❸ 函数的参数部分比较复杂，所以先解析这部分。同样地，先着眼于标识符。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function(func is) returning

❹ 因为有括号，所以这里解释*。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function(func is pointer to) returning

❺ 解释用于函数的()。这里的参数还是比较简单的，是 void（无参数）。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function(func is pointer to function (void) returning) returning

❻ 解释类型指定符 void。这样就结束了 atexit 的参数部分的解释。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function(func is pointer to function(void) returning void) returning

❼ 解释数据类型修饰符 int。

```
int atexit(void (*func)(void));
```

英语的表达为：

atexit is function (func is pointer to function (void) returning void) returning int

❽ 翻译成中文……

atexit 是返回 int 的函数（参数是，指向返回 void 没有参数的函数的指针）。

下面是一个更加复杂的例子。

标准库中有一个 signal()函数，它的原型声明如下，

```
void (*signal(int sig, void (*func)(int)))(int);
```

❶ 首先着眼于标识符。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is

❷ 相比*，()的优先顺序更高，所以先解释这部分。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function() returning

❸ 解释参数部分。这里有两个参数，第一参数是 int sig。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int,) returning

❹ 着眼另外一个参数。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is) returning

❺ 因为有括号，所以这里解释*。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to) returning

❻ 解释表示函数的()，参数为 int。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to function(int) returning) returning

❼ 解释数据类型修饰符 void。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to function(int) returning void) returning

❽ 参数部分已经解释结束。接着因为有括号，所以这里解释*。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to function(int) returning void) returning pointer to

❾ 解释表示函数的()，参数为 int。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to function(int) returning void) returning pointer to function(int) returning

❿ 最后，添上 void。

```
void (*signal(int sig, void (*func)(int)))(int);
```

英语的表达为：

signal is function(sig is int, func is pointer to function(int) returning void) returning pointer to function(int) returning void

⓫ 翻译成中文……

signal 是返回"指向返回 void 参数为 int 的函数的指针"的函数，它有两个参数，一个是 int，另一个是"指向返回 void 参数为 int 的函数的指针"。

如果能读懂这种难度的声明，我想应该不会再有什么让你畏惧的 C 声明了。

下面的说明可能会让你对 C 语言感到更加不快。

signal()是用于注册信号处理（当中断发生时被调用的函数）的函数。此函数的返回值是之前注册的处理当前信号中断的函数。

也就是说，其中的一个参数和返回值，它们都是相同的类型——指向信号处理函数的指针。在一般的语言中，同样的表现模式出现两次并不会让你感到不适，但是解释 C 语言声明的过程是"一会儿向左一会儿向右"，因此，表示返回值的部分散落了在左右两侧。

此时，运用 typedef 可以让声明变得格外得简洁。

```
/*摘录于 FreeBSD 的 man page */
typedef void(*sig_t)(int);

sig_t signal(int sig, sig_t func);
```

sig_t 代表"指向信号处理函数的指针"这个类型。

3.6 应该记住：数组和指针是不同的事物

3.6.1 为什么会引起混乱

首先，请允许我强调一下本章的重要观点。

C 语言的数组和指针是完全不同的。

大家都说 C 语言的指针比较难，可是真正地让初学者"挠墙"的，并不是指针自身的使用，而是"混淆了数组和指针"。此外，很多"坑爹"的入门书对指针和数组的讲解也是极其混乱。

比如，*K&R* 中就有下面一段文字（p.119），

> C 语言的指针和数组之间有很强的关联关系，因此必须将指针和数组放在一起讨论。

很多 C 程序员认为"数组和指针是几乎相同的事物"，这种认识是引起 C 的混乱的主要原因。

从图 3-17 中可以一目了然地看出，数组是一些对象排列后形成的，指针则表示指向某处。它们是完全不同的。

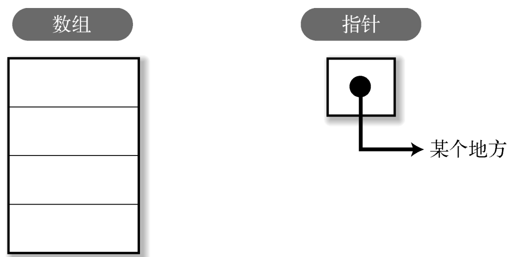


图 3-17　数组和指针

带着“数组和指针是几乎相同的事物”这样的误解，初学者经常写出下面这样的代码：

```
int *p;

p[3] = ……  ←— 突然使用没有指向内存区域的指针
```

——自动变量的指针在初期状态，值是不定的。

```
char str[10];
    ⋮
str = "abc";    ←— 突然向数组赋值
```

——数组既不是标量，也不是结构体，不能临时使用。

```
int p[];   ←— 使用空的[]声明局部变量
```

——只有在“函数的形参的声明”中，数组的声明才可以被解读成指针。

对于数组和指针，它们在哪些地方是相似的，又在哪些地方是不同的——不好意思，可能在下面会出现和前面重复的内容。

3.6.2　表达式之中

在表达式中，数组可以被解读成指向其初始元素的指针。所以，可以写成下面这样：

```
int *p;
int array[10];

p = array;     ←— 将指向 array[0]的指针赋予 p
```

可是，反过来写成下面这样：

```
array = p;
```

就是不可以的。确实，在表达式中 array 可以被解读成指针，可是，本质上它其实是被解释成了&array[0]，此时的指针是一个**右值***。

* 此时的指针是右值这个理由之外，在标准中，数组也不是“可变更的左值”。

比如，对于 int 类型的变量 a，a = 10;这样的赋值是可以的，但肯定没有人想做 a + 1 = 10;这样的赋值吧。尽管 a 和 a + 1 都是 int，但是 a + 1 没有对应的内存区域，只是一个右值，所以不能被赋值。同样的道理，array 也不能被赋值。

此外，对于下面这个指针，

```
int *p;
```

如果 p 指向了某个数组，自然可以通过 p[i]的方式进行访问，但这并不代表 p 就是数组。

p[i]只不过是*(p + i)的语法糖，只要 p 正确地指向一个数组，就可以通过 p[i]对数组的内容进行访问，就像图 3-18 表现的这样。

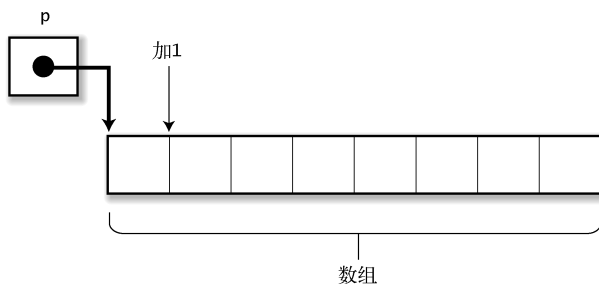


图 3-18　使用指针访问数组

如果是“指针的数组”和“数组的数组”，就会有很大的不同。

```
char *color_name[] = {          ←— 指针的数组
    "red",
    "green",
    "blue",
};
```

对以上的代码进行图解（参照图 3-19），

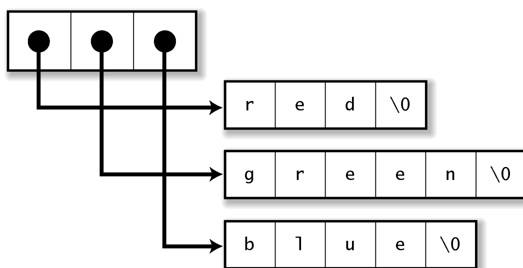


图 3-19　指针的数组

```
char color_name[][6] = {    ←— 数组的数组
    "red",
    "green",
    "blue",
};
```

对以上的代码进行图解（参照图 3-20），

r	e	d	\0	\0	\0	g	r	e	e	n	\0	b	l	u	e	\0	\0

图 3-20　数组的数组

以上两种情况都可以用 `color_name[i][j]`的方式对数组进行访问，但是内存中数据的布局是完全不同的。

3.6.3　声明

只有在声明函数的形参的时候，数组的声明才能解读成指针的声明（参照 3.5.1 节）。

以上的语法糖，与其说使 C 变得更加容易理解，倒不如说它使 C 语言的语法变得更加混乱。是不是有很多人这么想？我就是其中的一个*。而且 *K&R* 的说明更是使这种混乱局面雪上加霜。

* 虽然使用这个语法糖可以让多维数组作为参数被传递时更容易理解……

在不是声明函数的形参的时候，数组声明和指针的声明是**不可能相等**的。

使用 `extern` 的时候是最容易出现问题的（参照 3.5.2 节）。另外，声明局部变量或者结构体的成员时，写成

```
int hoge[];
```

会引起语法错误*。

* 对于结构体的成员，在 ISO C99 中是允许这种写法的。

存在数组初始化表达式的情况下，可以使用空的`[]`，但这是因为编译器能够计算出数组元素的个数，所以可以省略书写元素个数。仅此而已，这种特征和数组扯不上任何关系。

> 要　点
> **【非常重要!!】**
> **数组和指针是不同的事物。**

第 4 章

数组和指针的常用方法

4.1 基本的使用方法

4.1.1 以函数返回值之外的方式来返回值

这种手法其实已经在 1.2.6 节中说明过，本节让我们回头再总结一下。

在 C 中，可以通过函数返回值。可是，函数只能返回一个值。

在大型的程序中，经常需要通过返回值返回程序处理的状态（比如是否成功，如果失败，还需要返回失败的原因）。

如果将指针作为参数传递给函数，此后在函数内部对指针指向的对象填充内容，就可以从函数返回多个值。

此时，假设需要返回的数据的类型为 T，参数的类型就成为“指向 T 的指针”。

代码清单 4-1 中，将指向 `int` 和 `double` 的指针传递给函数，此后在函数内部对这两个指针指向的变量设定值。

> 要　点
> **如果需要通过函数返回值以外的方式返回值，将“指向 T 的指针”（如果想要返回的值的类型为 T）作为参数传递给函数。**

代码清单 4-1　output_argument.c

```
 1:  #include <stdio.h>
 2:
 3:  void func(int *a, double *b)
 4:  {
 5:      *a = 5;
 6:      *b = 3.5;
 7:  }
 8:
 9:  int main(void)
10:  {
11:      int    a;
12:      double b;
13:
14:      func(&a, &b);
15:      printf("a..%d b..%f\n", a, b);
16:
17:      return 0;
18:  }
```

4.1.2　将数组作为函数的参数传递

本节的内容在 1.3.6 节中也进行过说明，这里再做一次总结。

在 C 语言中，数组是**不能**作为参数进行传递的。但可以通过传递指向数组初始元素的指针，使得在函数内部操作数组成为可能。

因此，在函数这一侧，通过

```
array[i]
```

这种方式，就可以引用数组的内容。因为在本质上，array[i]只不过是*(array + i)的语法糖。

代码清单 4-2 中，将数组 array 传递给 func()，之后在 func()内部将 array 的内容输出。

func()还以参数 size 来接收数组 array 的元素个数。这是因为 array 只是一个指针，所以 **func()并不知道调用方数组的元素个数**。

main()中 array 的类型是“int 的数组”，因此，在 16 行可以用 sizeof 运算符取得数组元素的个数。

可是，在 func()中，参数 array 的类型是“指向 int 的指针”，即使使用 sizeof(array)，取出来的也只是指针自身的大小。

当然，我们可以模仿字符串的做法——在数组的末尾都加上'\0'，通过在函数内部检索'\0'，就可以计算出字符串中的字符个数。

代码清单 4-2　将数组作为参数传递

```
 1:  #include <stdio.h>
 2:
 3:  void func(int *array, int size)
 4:  {
 5:      int      i;
 6:
 7:      for (i = 0; i < size; i++) {
 8:         printf("array[%d]..%d\n", i, array[i]);
 9:      }
10:  }
11:
12:  int main(void)
13:  {
14:      int array[] = {1, 2, 3, 4, 5};
15:
16:      func(array, sizeof(array) / sizeof(int));
17:
18:      return 0;
19:  }
```

> 要　点
> **想要将类型 T 的数组作为参数进行传递，可以考虑传递“指向 T 的指针”。可是，作为被调用方是不知道数组的元素个数的，所以在必要的情况下，需要使用其他方式进行参数传递。**

4.1.3　可变长数组

一般情况下，C 语言在编译时必须知道数组的元素个数，但是也可以使用 malloc()在运行时再为数组申请必要的内存区域。

这种数组，在本书中被称为**可变长数组***。

代码清单 4-3 中，首先让用户输入需要的内存大小（11 ~ 13 行），在第 15 行使用 malloc()分配数组所需的内存区域（省略对返回值的检查）。

第 17 ~ 19 行，给数组赋值，并且在 20 ~ 22 行输出数组的内容。

* 虽然我们经常这么叫，但是不能因此就认为这是最常见的称呼。

代码清单 4-3　variable_array.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main(void)
 5:  {
 6:      char        buf[256];
 7:      int         size;
 8:      int         *variable_array;
 9:      int         i;
10:
11:      printf("Input array size>");
12:      fgets(buf, 256, stdin);
13:      sscanf(buf, "%d", &size);
14:
15:      variable_array = malloc(sizeof(int) * size);
16:
17:      for (i = 0; i < size; i++) {
18:          variable_array[i] = i;
19:      }
20:      for (i = 0; i < size; i++) {
21:          printf("variable_array[%d]..%d\n", i, variable_array[i]);
22:      }
23:
24:      return 0;
25:  }
```

如果想要修改已经分配了的可变长数组的大小，你可以使用 `realloc()`。

代码清单 4-4 中，每当用户输入一个 `int` 类型的值，程序都会使用 `realloc()` 扩展 `variable_array` 的内存区域（这里也省略了对返回值的检查）。

代码清单 4-4　realloc.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main(void)
 5:  {
 6:      int         *variable_array = NULL;
 7:      int         size = 0;
 8:      char        buf[256];
 9:      int         i;
10:
11:      while (fgets(buf, 256, stdin) != NULL) {
12:          size++;
13:          variable_array = realloc(variable_array, sizeof(int) * size);
14:          sscanf(buf, "%d", &variable_array[size-1]);
15:      }
16:
```

```
17:     for (i = 0; i < size; i++) {
18:         printf("variable_array[%d]..%d\n", i, variable_array[i]);
19:     }
20:
21:     return 0;
22: }
```

必须要引起注意的是，在使用 malloc()实现可变长数组的时候，程序员必须自己来管理数组的元素个数。

这和在将数组作为参数进行传递时，被调用方无法知道数组长度的理由一样——malloc()得到的不是数组，而是指针。

> 要 点
> **在需要获得类型 T 的可变长数组时，可以使用 malloc()来动态地给“指向 T 的指针”分配内存区域。**
> **但此时需要程序员自己对数组的元素个数进行管理。**



补充 Java 的数组

本书是 C 的参考书，在这里提到 Java 的数组只是给大家做一个参考。

Java 中的数组只能使用内存堆区域。在 C 的函数中写成下面这样：

```
int hoge[10];
```

在大多数处理环境中，数组本身是分配在栈中的。Java 不能写成上面这样，而应该写成下面这样：

```
int[] hoge = new int[10];
```

new 相当于 C 的 malloc()，上面的 Java 语句和下面的 C 语句具有几乎相同的意义，

```
int *hoge = malloc(sizeof(int) * 10);
```

因此，在 Java 中经常使用指针引用数组。比如，使用下面的方式进行数组的赋值：

```
int[] hoge = new int[10];
int[] piyo = hoge;
```

如图 4-1 所示，这里的 hoge 和 piyo 都是指向数组实体的指针变量。

可是，Java 的数组和在 C 中使用 malloc()分配的数组，有决定性的不同：Java 的数组是知道自身的长度的。因此，对于

```
int[] hoge = new int[10];
```

使用 hoge.length，可以知道数组的长度为 10*。

* 这里为什么不是 hoge.length()，也是 Java 的一个谜团，明明可以通过 length()获得 String 的长度。

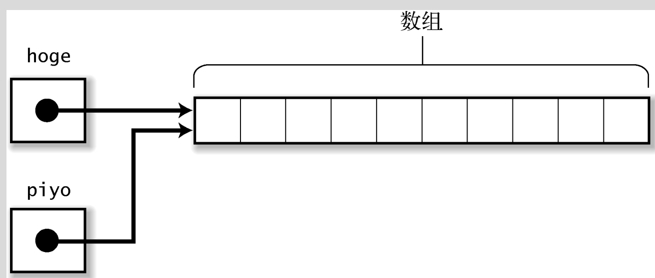


图 4-1 将 Java 的数组赋给变量

此外，尽管 Java 的数组是保存在堆中的，但却不能改变长度，它没有类似于 C 的 realloc()这样的函数。在很多 Java 语法的疑团中，这一点尤其让我感到费解*。真是太不方便了*！

* 倘若要实现 array.setSize(newSize) 这样的功能，一旦调用 realloc()，地址就会发生变化，因此，通过使用指针或者句柄进行间接引用，来实现 VM 以外的 VM，是一件难度很高的事情……莫非是因为这个原因？

* 特别是在通用性不太好的 Java 中，集合类库（Collection Library）提供的功能并不能完全满足需要。

4.2 组合使用

4.2.1 可变长数组的数组

让我们考虑开发一个管理“今天的标语”的程序。

周一的标语是“日行一善”，周二的标语“常回家看看”，等等。哈哈，是不是太说教了？

一周有 7 天，这是不会变化的。但是，标语的字数是相互不同的。对于“勿以善小而不为”，中文[①]的一个汉字为 2 个字节，末尾加上'\0'，就是 15 个字节；对于“常回家看看”，就是 11 个字节。此时，如果按照最长标语的字数声明一个二维数组，就避免不了浪费一些内存。

① 原文这里是“日语汉字”，日语汉字的一个字的长度也是 2 个字节。——译者注

此外，为了让用户可以自由地修改每天的标语，会考虑使用配置文件*，所以标语的最大长度我们是无法预测的。

* 一般也是这样的吧。

因为标语的长度是可变的，使用“char 的可变长数组”应该是一个很好的选择。也就是说，一周的标语可以放在“char 的可变长数组的数组（元素个数 7）”中（参照图 4-2）。

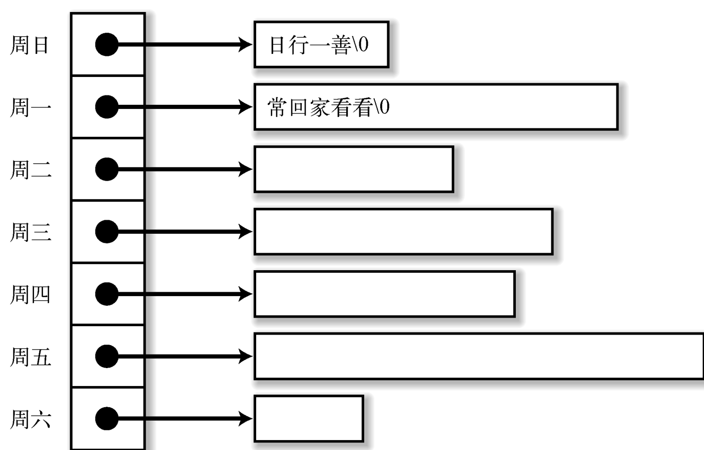


图 4-2　一周的标语

可以像下面这样进行声明：

```
char *slogan[7];
```

实现可变长数组的时候，一般需要开发者自己管理元素的个数。但是此时保存的是字符串，字符串必然是使用空字符结束的，所以不需要保持元素个数（在需要知道元素个数时，可以计算获得）。

如果从配置文件中读取一周的标语，程序应该就像代码清单 4-5 这样（这里省略了对 malloc()返回值的检查）。

代码清单 4-5　read_slogan.c

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #include <string.h>
4:  void read_slogan(FILE *fp, char **slogan)
5:  {
6:      char buf[1024];
7:      int  i;
```

```
 8:
 9:      for (i = 0; i < 7; i++) {
10:          fgets(buf, 1024, fp);
11:
12:          /*删除换行字符*/
13:          buf[strlen(buf)-1] = '\0';
14:
15:          /*分配保存一个标语的内存空间*/
16:        slogan[i] = malloc(sizeof(char) * (strlen(buf) + 1));
17:
18:          /*复制标语的内容*/
19:          strcpy(slogan[i], buf);
20:      }
21:  }
22:
23:  int main(void)
24:  {
25:      char *slogan[7];
26:      int i;
27:
28:      read_slogan(stdin, slogan);
29:
30:      /*输出读取的标语*/
31:      for (i = 0; i < 7; i++) {
32:          printf("%s\n", slogan[i]);
33:      }
34:
35:      return 0;
36:  }
```

代码清单 4-5 中，从标准输入读取一周的标语，之后再将其输出（第 31 ~ 33 行）。

在程序中，通过 `slogan[i]`可以取出指向每条标语的初始字符的指针，如果要取出标语第 *n* 个字符，就写成下面这样*：

* 汉字都是 2 个字符，所以通过“第 *n* 个字符”可能是取不出某个汉字的。这里姑且让我们先忽视掉这个问题吧☺。请暂时把它们当成英语的标语。

```
slogan[i][n]
```

`slogan` 不是多维数组（数组的数组），其内存布局完全不同于多维数组。

对于下面的多维数组的声明，

```
int hoge[10][10];
```

`hoge[i]`的类型是“`int` 的数组(元素个数 10)”。因为在表达式中数组可以解读成指针，`hoge[i]`就成为“指向 `int` 的指针”，所以可以通过 `hoge[i][j]`引用到数组的内容。

对于 `slogan`，`slogan[i]`从一开始就是指针，所以还是可以写成

slogan[i][n]这样。

此外，通过第 6 行的代码，我们能够发现标语的最大长度限制为 1024 个字节。

> 既然是“char 的可变长数组”，为什么不取消对标语长度的限制呢？如果无论如何都要添加限制，那么当初用多维数组就好了……

抱有上面想法的人请做一下下面的思考。对于多维数组，如果同样也加上“最大 1024 个字符”的限制，声明如下，

```
char slogan[7][1024];
```

此时，内存消耗为 7 × 1024 个字符。可是在代码清单 4-5 中，读取 1024 个字符时使用一个临时缓冲区就解决问题了。而且，由于这个数组是自动变量，所以，read_slogan()执行结束时这个缓冲区就会释放*。这种方式下，尽管有字符数的限制。但是这个限制是非常宽松的，从使用效果上来看，这个方式还是非常实用的。

可是在某些情况下，这种对字符数做出的限制，还是会让人感觉不便*，此时，可以使用 malloc()动态分配读取字符用的临时缓冲区。如果空间不足，再考虑通过 realloc()进行内存区域扩展。

代码清单 4-6 就是实现了可以读取任意长度的行的例程。

* 也有一些处理环境中栈空间的大小是固定的，而且也非常小。这种情况下，如果使用自动变量声明特别大的数组，有时会发生栈溢出问题。

* 顺便介绍一下，GNU 的编程标准中，取消了对这种方式的限制（ http://www.sra.co.jp/public/sra/product/wingnut/standards-j.html ）。

代码清单 4-6　read_line.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <assert.h>
 4:  #include <string.h>
 5:  #define ALLOC_SIZE      (256)
 6:
 7:  /*
 8:   *读取行的缓冲，必要时进行扩展。但是区域不会被缩小。
 9:   *调用 free_buffer()释放。
10:   */
11:  static char *st_line_buffer = NULL;
12:
13:  /*
14:   *在 st_line_buffer 前方被分配的内存区域的大小。
15:   */
16:  static int  st_current_buffer_size = 0;
17:
18:  /*
19:   * st_line_buffer 中现在保存的字符的大小。
```

```
20:  */
21: static int  st_current_used_size = 0;
22:
23: /*
24:  *如有必要，扩展 st_line_buffer 前方的内存区域。
25:  *在 st_line_buffer 末尾追加一个字符。
26:  */
27: static void
28: add_character(int ch)
29: {
30:     /*
31:      *此函数每次被调用，st_current_used_size 都必定会增加 1，
32:      *正常的情况下，下面的断言肯定不会出错。
33:      */
34:     assert(st_current_buffer_size >= st_current_used_size);
35:
36:     /*
37:      * st_current_used_size 达到 st_current_buffer_size 的时候，
38:      *扩展缓冲区的内存区域。
39:      */
40:     if (st_current_buffer_size == st_current_used_size) {
41:         st_line_buffer = realloc(st_line_buffer,
42:                                  (st_current_buffer_size + ALLOC_SIZE)
43:                                        * sizeof(char));
44:         st_current_buffer_size += ALLOC_SIZE;
45:     }
46:     /*在缓冲区末尾追加一个字符*/
47:     st_line_buffer[st_current_used_size] = ch;
48:     st_current_used_size++;
49: }
50:
51: /*
52:  *从 fp 读取一行字符，一旦读到文件末尾，就返回 NULL。
53:  */
54: char *read_line(FILE *fp)
55: {
56:     int         ch;
57:     char        *ret;
58:
59:     st_current_used_size = 0;
60:     while ((ch = getc(fp)) != EOF) {
61:         if (ch == '\n') {
62:             add_character('\0');
63:             break;
64:         }
65:         add_character(ch);
66:     }
67:     if (ch == EOF) {
68:         if (st_current_used_size > 0) {
69:             /*如果最终行后面没有换行*/
70:             add_character('\0');
```

```
71:         } else {
72:             return NULL;
73:         }
74:     }
75:
76:     ret = malloc(sizeof(char) * st_current_used_size);
77:     strcpy(ret, st_line_buffer);
78:
79:     return ret;
80: }
81:
82: /*
83:  *释放缓冲区内存。其实即使不调用这个函数也不会有什么问题，
84:  *但对于那些抱有“程序结束时，最好使用free()释放掉malloc()分配的内存
      区域”这种想法的人，
85:  *可以调用这个函数。
86:  */
87: void free_buffer(void)
88: {
89:     free(st_line_buffer);
90:     st_line_buffer = NULL;
91:     st_current_buffer_size = 0;
92:     st_current_used_size = 0;
93: }
```

代码清单 4-7 read_line.h

```
1: #ifndef READ_LINE_H_INCLUDED
2: #define READ_LINE_H_INCLUDED
3:
4: #include <stdio.h>
5:
6: char *read_line(FILE *fp);
7: void free_buffer(void);
8:
9: #endif /* READ_LINE_H_INCLUDED */
```

read_line()将读取的一行字符作为返回值返回（删除了换行字符）。如果读到了文件末尾，返回 NULL。

在 read_line()中，指针 st_line_buffer*指向缓冲区的初始位置，该缓冲区用于存放临时读取的字符。当缓冲区空间不足时，缓冲区会被扩展 ALLOC_SIZE 大小的区域。其实使用这种方式，由于非常频繁地调用 realloc()，会降低程序运行效率，同时也带来内存碎片化的风险（参照 2.6.5 节）。

* 对于生命周期为“文件内”的 static 变量，我一般习惯加上前缀 st_。

一旦读到行末，该函数会跟据当前行的大小重新分配内存区域（第 76 行），

然后将 st_line_buffer 的内容复制到此内存区域。因为下一次的调用还会使用缓冲区，所以此时无需释放 st_line_buffer*。

* 这种方法中使用了 static 变量，所以不是可重入的（reentrant）。多线程的情况下，也会出现问题。只要不在这些情况下，这种方式还是很有效的。

因为 st_line_buffer 只会伸长不会缩短，所以每次 st_line_buffer 只会消费至今为止读取的最长的行的大小（+α）。不管怎么说也就是这一个内存区域用于缓冲，所以不去管它也不会出什么问题。对于抱有“程序结束时，最好使用 free()释放掉 malloc()分配的内存区域”这种想法的人，可以在最后调用 free_buffer。

read_line()中，因为通过 malloc()分配了字符串所需要的内存区域，所以一旦使用结束必须在调用方使用 free()释放所分配的内存区域。

```
char *str;

str = read_line(fp);

/*一系列处理*/

free(str);      ←——一旦终止使用就释放!
```

此外，我们在代码清单 4-6 中省略了对返回值的检查。其实对于可以通用的函数，应该切实做好返回值的检查工作。因此，在 4.2.4 节中提供了这方面的例程。

4.2.2 可变长数组的可变长数组

4.2.1 节中使用可变长数组来表现一个标语，但标语的个数固定为一周的天数（7 个）。

如果需要在内存中加载任意行数的文本文件，可以考虑使用“可变长数组的可变长数组”（参照图 4-3）。

“类型 T 的可变长数组”是通过“指向 T 的指针”来实现的（但是元素个数就需要自己来管理）。

因此，如果需要“T 的可变长数组的可变长数组”，可以使用“指向 T 的指针的指针”（参照图 4-3）。

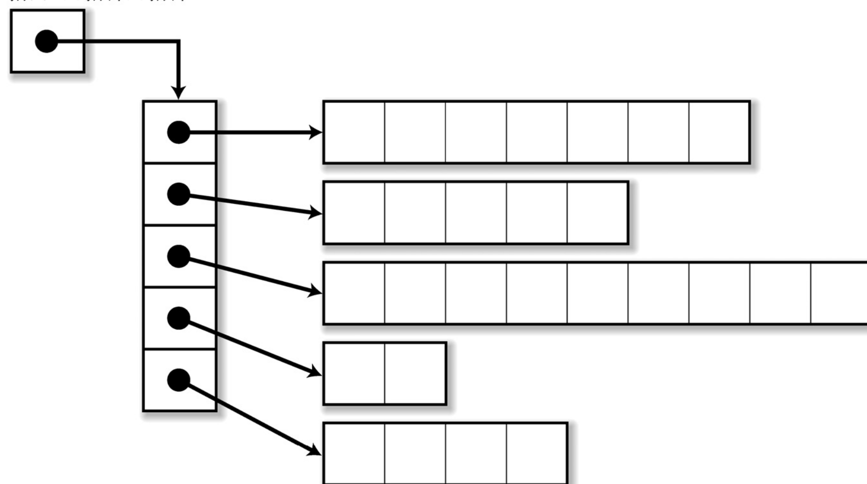


图 4-3　可变长数组的可变长数组

代码清单 4-8 中，从标准输入读取文本文件的内容，并且将其用于标准输出。为了读取任意长度的行，这里使用了代码清单 4-6 的函数 `read_line()`。

代码清单 4-8　read_file.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <assert.h>
 4:
 5:  #define ALLOC_SIZE      (256)
 6:
 7:  #include "read_line.h"
 8:
 9:
10:
11:  char **add_line(char **text_data, char *line,
12:                  int *line_alloc_num, int *line_num)
13:  {
14:      assert(*line_alloc_num >= *line_num);
15:      if (*line_alloc_num == *line_num) {
16:          text_data = realloc(text_data,
17:                              (*line_alloc_num + ALLOC_SIZE)
                                 * sizeof(char*));
18:          *line_alloc_num += ALLOC_SIZE;
19:      }
20:      text_data[*line_num] = line;
21:      (*line_num)++;
22:
23:      return text_data;
24:  }
```

```
25:
26:  char **read_file(FILE *fp, int *line_num_p)
27:  {
28:      char        **text_data = NULL;
29:      int         line_num = 0;
30:      int         line_alloc_num = 0;
31:      char        *line;
32:
33:      while ((line = read_line(fp)) != NULL) {
34:          text_data = add_line(text_data, line,
35:                               &line_alloc_num, &line_num);
36:      }
37:      /*将 text_data 缩小到实际需要的大小 */
38:      text_data = realloc(text_data, line_num * sizeof(char*));
39:      *line_num_p = line_num;
40:
41:      return text_data;
42:  }
43:
44:  int main(void)
45:  {
46:      char        **text_data;
47:      int         line_num;
48:      int         i;
49:
50:      text_data = read_file(stdin, &line_num);
51:
52:      for (i = 0; i < line_num; i++) {
53:          printf("%s\n", text_data[i]);
54:      }
55:
56:      return 0;
57:  }
```

因为不读到文件的最后就无法知道总共的行数，所以在 read_file()中，对于指针数组也使用 realloc()顺序地将其占用的内存空间加长。

read_line()中，为了共享一些变量，使用了文件内的 static 变量。但这里使用了通过参数传递指针的方式。通过文件内的 static 变量和全局变量共享数据时，因为无法知道“值在什么地方被改写”，所以在很多情况下通过参数传递指针的方式显得很有效。

4.2.3　命令行参数

正如 1.2 节中说明的那样，对于 main 函数，标准中指出必须要写成下面两种形式的中的一种*。

* 很多 UNIX 环境将指向环境变量的指针作为第 3 个参数传递，其实这并不符合标准。

```
int main(void)     ←— ①
```

或者

```
int main(int argc, char *argv[])    ←— ②
```

直到现在，我们还一直使用第①种形式，如果使用第②种形式，还可以取得命令行的参数。

比如，UNIX 中的 cat 命令——它用于输出文件的内容*。

像下面这样：

```
cat hoge.txt
```

命令名后是想要输出的文件名。如果像下面这样将多个文件名作为参数排列起来：

```
cat hoge.txt piyo.txt
```

输出就是将 hoge.txt 和 piyo.txt 两个文件的内容连接后所得的结果*。

其实，事先是无法知道 cat 有几个参数的。不仅如此，各参数（文件名）所对应文件的长度也无法预测。因此，这些参数可以使用“char 的可变长数组的可变长数组”、“指向指针的指针”来表现。

对于刚才的 main 函数的第②种形式，它其实和下面这种形式是完全一样的，

```
int main(int argc, char **argv)
```

你既然读到这了，我想应该能明白这是为什么*。

argv 在内存中的结构，如图 4-4 所示。

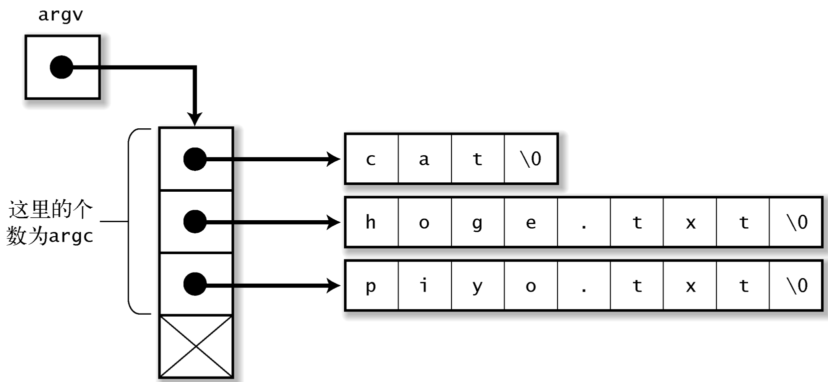


图 4-4　argvz 的结构

* DOS 中使用 type 命令。但是，DOS 的 type 命令不具备文件的连接功能（可以使用 copy 命令）。

* cat 是 concatenate（连接）的简写。

* 不理解的同学，请再次阅读 3.5.1 节。

argv[0]中保存了命令名自身。在程序输出错误提示信息时[*]，或者需要通过命令名称改变程序的行为时，会经常使用 argv[0]。

* UNIX 中，可以通过管道将命令连接起来执行处理，在错误信息中，为了在错误信息中提示某个命令自身的信息，需要给自己定义一个名称。

argc 中保存了参数的个数（包含了 argv[0]）。实际上，从 ANSI C 之后，会保证 argv[argc]肯定为 NULL，所以**完全可以没有** argc，但是目前仍然还有很多人习惯性地引用 argc。

代码清单 4-9 是 UNIX 的 cat 的简单实现。

和 UNIX 的 cat 一样，如果不指定参数，则使用标准输入（第 14 ~ 15 行）。

第 20~27 行，在 for 循环中顺序地处理参数指定的文件名。

世间有很多人在此时，一直顽固地拒绝使用循环计数，他们宁愿一边对 argc 做减法运算，一边前移 argv。我还是习惯使用计数器，然后使用下标访问，因为这种方式很容易让人理解。

代码清单 4-9　cat.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  void type_one_file(FILE *fp)
 5:  {
 6:      int ch;
 7:      while ((ch = getc(fp)) != EOF) {
 8:          putchar(ch);
 9:      }
10:  }
11:
12:  int main(int argc, char **argv)
13:  {
14:      if (argc == 1) {
15:          type_one_file(stdin);
16:      } else {
17:          int     i;
18:          FILE    *fp;
19:
20:          for (i = 1; i < argc; i++) {
21:              fp = fopen(argv[i], "rb");
22:              if (fp == NULL) {
23:                  fprintf(stderr, "%s:%s can not open.\n",
                     argv[0], argv[i]);
24:                  exit(1);
25:              }
26:              type_one_file(fp);
27:              fclose(fp);
28:          }
```

```
29:      }
30:
31:      return 0;
32:  }
```

4.2.4　通过参数返回指针

4.2.1 节中使用函数 read_line()（参照代码清单 4-6），将读取的行作为返回值返回，如果到达文件的终点则返回 NULL。

可是，作为返回值的形式，read_line()返回的是通过 malloc()分配的内存区域。代码清单 4-6 中并没有对返回值做检查。如果真的想要让 read_line()成为通用的函数，就必须好好地对返回值做检查，并且能向调用方返回函数的处理状态。

对于 read_line()向调用方返回的处理状态，下面列出了几种可能状况：

❶ 正常地读取了 1 行。
❷ 读到了文件的末尾。
❸ 内存不足导致处理失败。

将这些状态用枚举类型来表示：

```
typedef enum {
    READ_LINE_SUCCESS,      /*正常地读取了 1 行*/
    READ_LINE_EOF,          /*读到了文件的末尾*/
    READ_LINE_OUT_OF_MEMORY /*内存不足导致处理失败*/
} ReadLineStatus;
```

作为向调用方返回处理状态的方式，考虑像下面这样通过参数返回：

```
char *read_line(FILE *fp, ReadLineStatus *status);
```

这种方案是非常正确的，但是也有不少软件项目选择坚持“应该通过返回值返回处理状态”的观点。

但是，如果将返回值用于表示处理状态，那么当前通过返回值返回的读取到的字符串，就必须通过参数返回。

如果使用参数返回类型 T，可以使用“指向 T 的指针”。目前我们想要返回类型“指向 char 的指针”，所以参数的类型就应该是“指向 char 的指针的指针”。

因此，函数的原型如下：

```
ReadLineStatus read_line(FILE *fp, char **line);
```

代码清单 4-10 为修订版的头文件，代码清单 4-11 是实现代码。

代码清单 4-10　read_line.h（修订版）

```
 1:  #ifndef READ_LINE_H_INCLUDED
 2:  #define READ_LINE_H_INCLUDED
 3:
 4:  #include <stdio.h>
 5:
 6:  typedef enum {
 7:      READ_LINE_SUCCESS,      /*正常地读取了1行*/
 8:      READ_LINE_EOF,          /*读到了文件的末尾*/
 9:      READ_LINE_OUT_OF_MEMORY /*内存不足导致处理失败*/
10:  } ReadLineStatus;
11:
12:  ReadLineStatus read_line(FILE *fp, char **line);
13:  void free_buffer(void);
14:
15:  #endif /* READ_LINE_H_INCLUDED */
```

代码清单 4-11　read_line.c（修订版）

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <assert.h>
 4:  #include <string.h>
 5:  #include "read_line.h"
 6:  #define ALLOC_SIZE      (256)
 7:
 8:  /*
 9:   *读取行的缓冲，必要时进行扩展。但是区域不会被缩小
10:   *调用free_buffer()释放。
11:   */
12:  static char *st_line_buffer = NULL;
13:
14:  /*
15:   *在st_line_buffer前方分配的内存区域大小。
16:   */
17:  static int  st_current_buffer_size = 0;
18:
19:  /*
20:   * st_line_buffer中现在被保存的字符的大小。
21:   */
22:  static int  st_current_used_size = 0;
23:
24:  /*
```

```
25:  *在 st_line_buffer 末尾追加一个字符。
26:  *如果有必要，扩展 st_line_buffer 前方的内存区域。
27:  */
28: static ReadLineStatus
29: add_character(int ch)
30: {
31:     /*
32:      *此函数每次被调用，st_current_used_size 都必定会增加 1。
33:      *正常的情况下，下面的断言肯定不会出错。
34:      */
35:     assert(st_current_buffer_size >= st_current_used_size);
36:
37:     /*
38:      *当 st_current_used_size 达到 st_current_buffer_size 的时候，
39:      *扩展缓冲区的内存区域。
40:      */
41:     if (st_current_buffer_size == st_current_used_size) {
42:         char *temp;
43:         temp = realloc(st_line_buffer,
44:                        (st_current_buffer_size + ALLOC_SIZE)
45:                           * sizeof(char));
46:         if (temp == NULL) {
47:             return READ_LINE_OUT_OF_MEMORY;
48:         }
49:         st_line_buffer = temp;
50:         st_current_buffer_size += ALLOC_SIZE;
51:     }
52:     /*在缓冲区末尾追加一个字符。*/
53:     st_line_buffer[st_current_used_size] = ch;
54:     st_current_used_size++;
55:
56:     return READ_LINE_SUCCESS;
57: }
58:
59: /*
60:  *释放缓冲区内存。其实即使不调用这个函数也不会有什么问题，
61:  *但对于那些抱有“程序结束时，最好使用 free()释放掉 malloc()分配的
      内存区域”这种想法的人，
62:  *可以调用这个函数。
63:  */
64: void free_buffer(void)
65: {
66:     free(st_line_buffer);
67:     st_line_buffer = NULL;
68:     st_current_buffer_size = 0;
69:     st_current_used_size = 0;
70: }
71:
72: /*
73:  *从 fp 读取一行字符，一旦读到文件末尾，就返回 NULL。
74:  */
75: ReadLineStatus read_line(FILE *fp, char **line)
```

```
 76:  {
 77:      int                   ch;
 78:      ReadLineStatus        status = READ_LINE_SUCCESS;
 79:
 80:      st_current_used_size = 0;
 81:      while ((ch = getc(fp)) != EOF) {
 82:          if (ch == '\n') {
 83:              status = add_character('\0');
 84:              if (status != READ_LINE_SUCCESS)
 85:                  goto FUNC_END;
 86:              break;
 87:          }
 88:          status = add_character(ch);
 89:          if (status != READ_LINE_SUCCESS)
 90:              goto FUNC_END;
 91:      }
 92:      if (ch == EOF) {
 93:          if (st_current_used_size > 0) {
 94:              /*如果最终行后面没有换行*/
 95:              status =add_character('\0');
 96:              if (status != READ_LINE_SUCCESS)
 97:                  goto FUNC_END;
 98:          } else {
 99:              status = READ_LINE_EOF;
100:              goto FUNC_END;
101:          }
102:      }
103:
104:      *line = malloc(sizeof(char) * st_current_used_size);
105:      if (*line == NULL) {
106:          status = READ_LINE_OUT_OF_MEMORY;
107:          goto FUNC_END;
108:      }
109:      strcpy(*line, st_line_buffer);
110:
111:    FUNC_END:
112:     if (status != READ_LINE_SUCCESS && status != READ_LINE_EOF) {
113:          free_buffer();
114:      }
115:      return status;
116:  }
```

在 read_line()中，一旦 malloc()返回 NULL，程序马上通过 goto 将处理转移到 FUNC_END。

在处理失败的情况下，FUNC_END 调用 free_buffer()来释放缓冲区占用的内存。一般在内存不足的时候 malloc()才会发生错误，所以释放掉这部分内存，多少能腾出一些空间来完成后面的处理（只是**可能**）。

世上有很多主张“禁用 goto”的教条主义者，但在这种异常处理中，如果不用 goto，反而会让程序变得更复杂*。

这里顺便插一段，曾经打响“goto 威胁论”第一枪的 Edsger W. Dijkstra 老师在事后说了这么一段话（《计算机程序设计艺术》[8] p.41），

> 对于我的禁用 goto 的这个极端片面的想法，还是希望大家不要掉进迷信的圈套。通过代码写法上的某个技巧去解决程序设计上的问题，就和企图创造一个新兴宗教一样让人感到不快。

* 如果是 C++和 Java 这种具备异常处理机制的语言，当然可以不使用 goto。在 C 中也有 setjmp()/longjmp()，使用它们也能达到相似的效果。

> 要　点
> **异常处理中使用 goto，反而可以让程序更加整洁。**

4.2.5 将多维数组作为函数的参数传递

在 C 中，其实**不存在**多维数组，看上去貌似多维数组的其实是“数组的数组”。

将类型 T 的数组作为参数进行传递时，可以传递“指向 T 的指针”（参照 4.1.2 节）。因此，如果想要将“数组的数组”作为参数进行传递，可以考虑传递“指向数组的指针”。

在代码清单 4-12 中，将 3 × 4 的二维数组传递给函数 func()，然后在 func()中将其内容输出。

代码清单 4-12 pass_2d_array.c

```
 1:  #include <stdio.h>
 2:
 3:  void func(int (*hoge)[3])
 4:  {
 5:      int i, j;
 6:
 7:      for (i = 0; i < 4; i++) {
 8:          for (j = 0; j < 3; j++) {
 9:              printf("%d, ", hoge[i][j]);
10:          }
11:          putchar('\n');
12:      }
13:  }
14:
15:  int main(void)
16:  {
17:      int hoge[][3] = {
18:          {1, 2, 3},
```

```
19:           {4, 5, 6},
20:           {7, 8, 9},
21:           {10, 11, 12},
22:       };
23:
24:       func(hoge);
25:
26:       return 0;
27:   }
```

4.2.6　数组的可变长数组

假设要开发一个支持多点折线（曲线）的画笔工具。

考虑使用可变长数组来表现多点折线中的“点”，并且使用“double 的数组（元素个数 2）”来记录一个“点”。

因此，多点折线就可以被定义成：

double 的数组（元素个数 2）的可变长数组

可以使用“指向类型 T 的指针”实现“类型 T 的可变长数组*”，所以上面的定义可以变化成

* 但是，元素个数需要通过别的方式管理。

指向 double 的数组（元素个数 2）的指针

因此，分配多点折线的内存区域，可以写成下面这样：

```
double (*polyline)[2]; ←── polyline 是指向 double 数组(元素个数 2)的指针

/* npoints 是构成多点折线的坐标的个数*/
polyline = malloc(sizeof(double[2]) * npoints);
```

如果感觉理解上有些困难，不妨通过下面的方式对“double 的数组（元素个数 2）”这部分进行类型定义：

```
typedef double Point[2];
```

此时，ployline 的声明和内存区域的申请可以写成下面这样：

```
Point *polyline;    ←── polyline 是指向 Point 的指针

polyline = malloc(sizeof(Point) * npoints);
```

我想这样就容易理解了吧。

无论使用上面的哪种方式，第 i 个点的 X 坐标都写成

```
polyline[i][0]
```

Y 坐标都写成

```
polyline[i][1]
```

但是本书却不推荐大家使用本小节中介绍的方法，理由在下一小节给大家说明。

4.2.7　纠结于“可变”之前，不妨考虑使用结构体

在 4.2.6 节中，使用了“指向数组的指针”来表现多点折线。

假设有 5 条多点折线，应该通过什么样的方式来管理呢？

“多点折线”是“指向 double 的数组（元素个数 2）的指针”（需要自己管理元素个数）。因此，如果是“多点折线的数组（元素个数 5）”，就可以解释为“指向 double 数组（元素个数 2）的指针的数组（元素个数 5）”，声明如下：

```
double (*polylines[5])[2];
```

你既然读到这了，应该可以很轻松地读懂上面的声明吧。但客观地说，这个声明仍然比较复杂。

此外，因为有 5 根折线，所以对于每根折线对应的元素个数 npoint（每根折线上的坐标个数），就需要声明为以下数组：

```
int npoints[5];
```

如果不是 5 根，而是将任意数量的“折线”作为参数来接收的函数原型，应该大致是下面这样：

```
func(int polyline_num, double (**polylines)[2], int *npoints);
```

可以定义一个 Point 类型：

```
typedef double Point[2];
```

函数原型就可以变成下面这样：

```
func(int polyline_num, Point **polylines, int *npoints);
```

如果顺便再定义下面的类型，

```
typedef Point *Polyline;
```

上面的函数原型就变成了下面这样：

```
func(int polyline_num, Polyline *polylines, int *npoints);
```

尽管使用 typedef 可以简化声明，但还是不能让这个声明变得更加容易理解。

但要说最不尽如人意的地方，那还是“需要程序员自己管理数组的元素个数”这件事吧。

索性像下面这样使用结构体来定义 Point 吧：

```
typedef struct {
    double      x;
    double      y;
} Point;
```

同样地，也可以用结构体定义 Polyline：

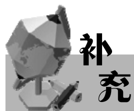
```
typedef struct {
    int         npoints;
    Point       *point;
} Polyline;
```

将 npoints 和 point 进行统一管理，编程工作好像变得更简单，并且也不需要做“X 坐标为[0]，Y 坐标为[1]”这样的暗喻。

在 CAD 这样的应用中，需要经常进行坐标的行列转换。对于 Point 这样的结构体，就要使用循环逻辑来回倒腾数据。此时不妨考虑使用下面的方式：

```
typedef struct {
    double      coordinate[3];
} Point;
```

其实，对于 2D 的画笔程序，具有 x，y 成员的结构体就已经够用了。



补充　什么是“宽度”可变的二维数组?

如图 4-5 所示，4.2.6 节中使用了“定长数组的可变长数组”。

但有时我们希望在运行时确定数组两个维度的大小。

很遗憾，C 语言不支持这种使用数组的方式。正如已经说明过的那样，所谓 C 语言的二维数组，其实本质上是“数组的数组”。为了引用数组的元素，C 语言必须在**编译时**就确定数组元素的类型大小。

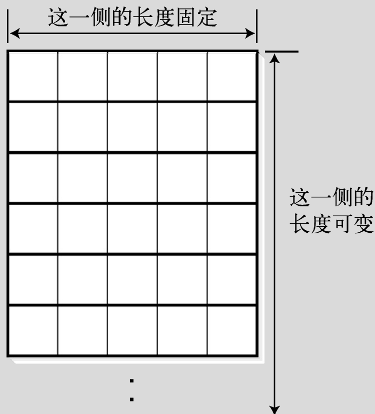


图 4-5　定长数组的可变长数组

如果使用 4.2.2 节中介绍的方法，当然可以实现如图 4-6 这样的“宽度可变的二维数组”的替代品。

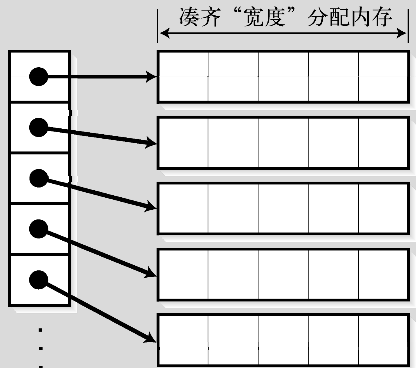


图 4-6　宽度可变的二维数组（替代品）实现 1

但由于这种方式需要多次调用 malloc()，所以在效率和速度上都不会令人满意。此外，过多的 malloc()调用，也会让 free()的调用变得很繁琐。

针对这个问题，可以将 malloc()的调用限制在两次，如图 4-7 这样将指针进行伸展。

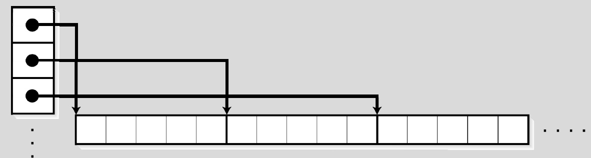


图 4-7　宽度可变的二维数组（替代品）实现 2

当然，无论是图 4-6 还是图 4-7，在引用数组的内容的时候都可以写成 array[i][j]这样的形式。

其实，现实中除了拘泥于 array[i][j]这样的写法，也是可以简单地使用一维可变长数组，使用 array[i * width + j]来引用数组的内容，这也许是最省事的方式了。



补充　Java 的多维数组

这部分可能有些离题，让我们来谈谈 Java。

4.1 节的补充内容“Java 的数组”中提到，Java 经常使用指针来操作数组。

此外，Java 和 C 一样不存在多维数组，也是使用数组的数组来实现多维数组（替代品）。但是，Java 和 C 不同的是，Java 的数组都是指针，所以所谓的“数组的数组”其实是“指向数组的指针的数组”。

在 Java 中，如果使用二维数组来表现“二维多点折线”，你可以写成下面这样：

```
// nPoints 为坐标的个数
double[][] polyline = new int[nPoints][2];
```

各元素在内存中的配置如图 4-8 所示。

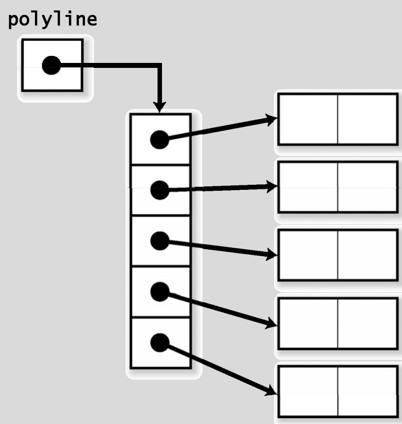


图 4-8　Java 的多点折线二维数组

你可以将 polyline[1]赋给 polyline[0]，也可以将 null 赋给 polyline[0]。

顺便介绍一下，Java 中的类（相当于 C 中的结构体）也是保存在堆中，

并且只能利用指针进行操作。因此，Java 中没有类似 C 中“结构体的数组”这样的对象。

在 4.2.7 节中，没有使用二维数组而是使用了结构体来表现多点折线。如果使用 Java 实现，其中各元素的内存布局如图 4-9 所示。

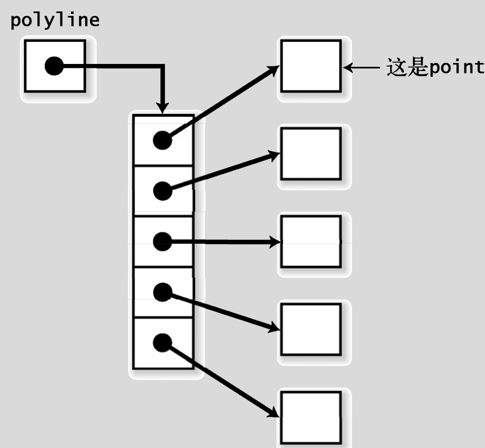


图 4-9 Java 的多点折线结构体（类）

可以使用下面的代码构造这样的结构：

```
// nPoints 为坐标的个数
Point[] polyline = new Point[nPoints];
for (int i = 0; i < nPoints; i++) {
    polyline[i] = new Point();
}
```

其实阻碍 Java 程序快速执行的最大原因就是 Java 过多地使用了堆操作*。

C++的对象可以不通过指针，而是通过实体来操作。为了实现这个特性，C++使用了带参数的构造方法以及继承的概念，导致了 C++编程变得非常复杂。

* 关于 Java 过多地进行堆操作这一点，可以说是面向对象语言的必然特征。

4.3 违反标准的技巧

4.3.1 可变长结构体

在 4.2.7 节中，使用了下面的方式定义了多点折线。

```
typedef struct {
    int     npoints;
    Point   *point;
} Polyline;
```

使用 malloc()动态地为 Polyline 分配内存的时候，需要调用两次 malloc()——两次在堆空间上分配内存（参照图 4-10）。

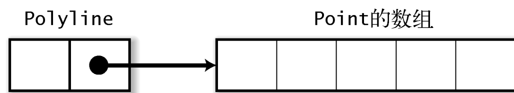


图 4-10 Polyline 的实现方法 1

从实现方法上来看，其实这样做也是非常正确的。但是，对于 malloc()分配的每个内存区域，通常都需要一些额外的管理区域。除此以外，内存碎片化也是一个问题（请参照第 2 章相关内容）。此时，不妨以下面的方式声明 Polyline 类型：

```
typedef struct {
    int     npoints;
    Point   point[1];
} Polyline;
```

并且使用下面的手法进行内存分配：

```
Polyline *polyline;

polyline = malloc(sizeof(Polyline) + sizeof(Point) * (npoints-1));
```

在这种方式下，如果使用 polyline->point[3]进行引用，会发生越界引用，因为 Polyline 类型的 point 成员的元素个数是 1。好在大部分的 C 处理环境都**不做数组范围检查**，你可以使用 malloc()在 Polyline 后面追加分配必要的内存空间（参照图 4-11）。

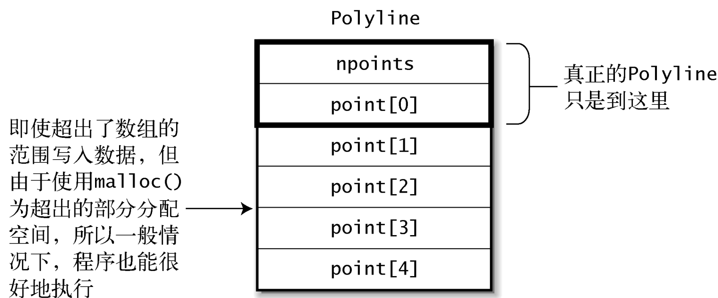


图 4-11 Polyline 的实现方法 2

通过这样的写法，**结构体的最后成员**不使用指针也可以直接保存可变长数组。这种（似乎）可以让结构体长度可变的技巧，称为“可变长结构体”（虽

然不是标准称呼）。

此外，如果将 Polyline 类型的成员 point 声明成 point[0]这样，在 malloc()的时候就不需要做 npoints - 1 这样的调整。但是 ANSI C 规定了数组的元素个数必须大于 0。虽然也有处理环境（如 gcc 等）允许声明元素个数为 0 的数组，但不管怎么说，这只是个别实现。

可是，可变长结构体这样的技巧并不总是有效的。比如，想要增加 Polyline 的坐标个数的时候，对于 Point 数组，可以通过 realloc()扩展必要的内存区域。但对于可变长结构体 Polyline，就需要整体重新分配内存区域，Polyline 自身的地址发生变化的可能性很高。如果有很多持有指向 Polyline 的指针的指针变量，必须对它们进行全部更新。这事儿还是挺麻烦的！

其实，“可变长结构体”这样的技巧，如果用于将结构体整体保存到文件，或者通过进程间通信、网络等条件进行传输，应该会产生不错的效果*。使用 fwrite()这样函数将内存中的结构输出的时候，指针类型的成员所指向的对象是输出不了的。但如果是可变长结构体，其内存区域自身是一个整体，因此可以简单地将所有数据输出。反过来，使用 fread()这样的函数，可以将数据整体再读取（还原）到内存*。

* 当然，正如 2.8 节中说明的那样，将结构体整体保存在文件中，然后通过网络进行传输这样的操作，可能自身会存在一些问题。但如果是通过同一台机器上的临时文件进行数据交换，或者在同一台机器上的进程间进行通信的情况下，即使使用了这种方法，也是没有问题的。

* Windows 的 BMP 文件就是通过这样的方式从内存中输出的。对于类似于 BMP 文件这样经常被用于多种环境的数据文件，利用“将整体结构体从内存中输出”的这种方式，无论如何都只是 Windows 特有的做法:-P

但无论怎样，从所谓的 ANSI C（ISO-IEC 9899-1990）定义的语法标准来看，可变长结构体还是属于**违反标准的技巧**。因为规范中指出，对于超出范围对数组的访问，并不保证其有效性和正确性。

毕竟大部分的环境都支持“可变长结构体”的手法，既然事实上已经被广泛使用，所以我们也就没有必要去故意回避。虽然提出了“严守规范编程”（strictly confirming program）的忠告，但在实际开发中也不能过于教条。

顺便告诉大家，ISO C99（ISO/IEC 9899:1999）中，可变长结构体已经被正式承认，并且成为专用语法。

4.3.2　从 1 开始的数组

C 的数组下标是从 0 开始的。初学者经常对此产生困惑，但是大部分情况下，“从 0 开始”要比“从 1 开始”让人感觉更加合适（参照 1.3 节的补充内容）。

但是，在对 FORTRAN 开发的程序进行移植的时候，情况就不是这样的了。

这种情况下，可以通过下面的方法勉强地做出从 1 开始的数组：

```
/*需要一个 1～10 的数组*/
double hoge_buf[10];
double *hoge = &hoge_buf[-1];
此时，hoge[1]恰好就指向 hoge_buf[0]。
```

如果是多维数组，就像下面这样：

```
/*需要 4×4 的二维数组*/
double a_buf[4][4];

double (*a)[4] = &a_buf[-1][-1];
```

其实很早之前，上面的手法就为大家所知，将 FORTRAN 程序转换成 C 程序的工具 f2c 中，就用到了这种手法。但是，严格地说，它**违反了 C 标准**。

C 语言中，对指向超出数组范围以外的指针，除非它指向“最后元素的下一个元素”，其他情形都属于**未定义**。

因此，对于下面的语句：

```
double *a = &a_buf[-1];
```

因为指针变量 a 指向了 a_buf[-1]，仅凭这一点就违反了标准，这和有没有使用当前这个指针进行实际地读写**无关**。如此这般让指针指向一个没有被分配的内存区域，在不同的处理环境中，CPU 可能会抛出异常，或者地址最后被“晕头转向”地指到了一个莫名其妙的地方。

可是，C 标准却偏偏承认指向数组“最后元素的下一个元素”的指针是合法的，究竟为什么会存在这个**不公平的潜规则**？关于这个问题，请阅读下面的补充内容。



补充　指针可以指到数组的最后元素的下一个元素

C 标准只承认指向数组的“最后元素的下一个元素”的指针是合法的。如果指针指向“最后元素的下下个元素”（这和有没有发生读写无关），此行为就被认定为未定义。

作为理由，Rationale 里记载了下面这样的一个例子：

```
SOMETYPE array[SPAN];
/* ... */
for (p = &array[0]; p < &array[SPAN]; p++)
```

这里没有使用循环计数器，而是使用指针遍历数组的各元素。

如此这般，当循环到达终点时，如果要问 p 指向哪里，答案当然是 &array[SPAN]了，也就是 array 的最后元素的下一个元素。

只是为了照顾到以前写的代码，标准才允许指针可以指到数组的“最后元素的下一个元素”。

话说回来，本书还是推荐“不要使用指针运算，而是使用下标来访问数组”。

如果只是为了照顾到很久以前的事，倒是大可不必勉强接受这个**奇怪的潜规则**[*]。

* 尽管在早期使用指针运算可以写出执行效率高的代码……

第 5 章

数据结构——真正的指针的使用方法

5.1 案例学习 1：计算单词的出现频率

5.1.1 案例的需求

直到第 4 章，本书主要对“声明的解读方法”和“数组和指针的微妙关系”进行了说明。

这些内容都属于 C 语言特有的话题，正因为如此，人们才普遍认为“C 的指针比较难”。

但是，指针同时也是构造链表或者树这样的“数据结构”必需的概念。因此，在比较成熟的、正统的编程语言中，必定存在指针*。本章给大家介绍在构造一般数据结构的过程中指针的使用方法。

* 对于链表这种程度的数据结构，如果有了集合类库，操作数据时就可以不用顾及指针的存在了。

下面是一个经常被使用的案例，非常对不起，这的确是一个没有新意、老掉牙的案例。

> 设计开发一个将文本文件作为输入，计算出其中各单词出现频率的程序。

这个程序的名称是 word_count*。

* 这里容易和 UNIX 的 wc 命令产生混淆，可以先不管它。

使用下面的命令行，

```
word_count 文件名
```

将英文文本文件的文件名作为参数传递给 word_count，程序执行的结果是：

将英文单词按照字母顺序排序后输出，并且在各单词的后面显示当前单词的出现次数。

如果省略参数，就将标准输入的内容作为输入进行处理。



补充

各种语言中对指针的叫法

正如我反复强调的那样，如果没有指针，就无法构造正统的数据结构，因此，比较成熟的、正统的编程语言，必定会存在指针*。

> 咦？我怎么听说 Java 就没有指针呢……

我可以负责任地告诉你，这是个**谣言**。

第 4 章的补充内容中也曾经提到，Java 只能通过指针来操作数组和对象，因此，Java 比 C 更离不开指针。

在早期的 Java 白皮书中，就有"Java 中没有指针"这样的说法*。Java 中被称为"引用"的概念，在 C 和 Pascal 的程序员看来，**怎么看都相当于指针**。我认为在"Java 中没有指针"这个观点的背后，弥漫着下面这样"狡猾的"市场营销的气味，

> 因为对于 C 语言，大家都认为"指针比较难"，如果强调"没有指针"，编程新手也许更容易接受。

但是 Java 的引用又和 C 的指针有着很大的不同。Java 没有指针运算，因此不存在指针运算和数组之间的那种微妙关系，此外你也不能取得指向变量的指针。如果你认为这些差别能成为"Java 中没有指针"的理由，那么 Pascal 是不是也没有指针呢？

除 Java 之外，Lisp、Smalltalk 和 Perl（Ver.5 以后）中相当于指针的对象也被称为"引用"，但是也有人会使用"指针"这样的叫法。也就是说，这些语言并没有严格地将"引用"和"指针"分开。因为它们的本质相同，所以 Java 故意强调"没有指针"，反而让人觉得奇怪*。

Ruby 中连字符串这样的基本类型也不是不可变的，像这样的语言"没有指针"，是不是很危险？

Pascal、Modula2/3 和 C 一样，都称之为指针。

Ada 中的名称为"Access 类型"。这种叫法有点人让人摸不着头脑。

悲哀的是，C++在语法上将"指针"和"引用"区别成两个不同的概念。

C++的"指针"和 C、Pascal 的"指针"，以及 Java 的"引用"同义。其次，C++中的"引用"是指本来应该被称为"别名"（`alias`）的对象，

* 以前，FORTRAN、COBOL 和 BASIC 中都没有指针，但是在 fortran90、Visual Basic 等升级版中，正式引入了指针功能。

* 请参照 http://java.sun.com/docs/overviews/java/java-overview-1.html。

* 诞生在日本的面向对象的脚本语言 Ruby，作者在自己的著作中就断言"Ruby 中没有指针这样的概念"，其实 Ruby 中也有叫做"引用"的指针。

正因为是别名，所以一旦确定“别名是什么”，就再也不能修改了。

实际上，C++的术语“引用”也是通过指针实现的，所以它其实是一个重复的功能。很多熟练的 C++程序员往往不使用“引用”，而总是使用指针。但是，在某些运算符重载，以及复制构造函数的场景下，可能会不得不使用“引用”。对于 C++，有人说它太深奥，有人说使用它开发项目成本太高，甚至有人质疑“是否存在理解 C++全貌的人”……总之，C++也是一门让人纠结的开发语言。

5.1.2 设计

在开发大型应用程序的时候，非常有必要将程序按照单元功能（模块）分开。尽管这次的程序很小，但为了达到练习的目的，我们准备将这次的程序模块化。

将 `word_count` 程序分割成如图 5-1 的样子。

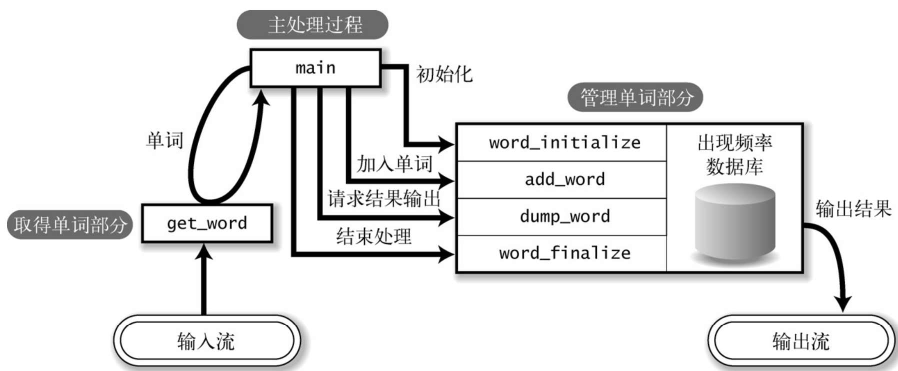


图 5-1 `word_count` 的模块结构

❶ 取得单词部分

从输入流（文件等）一个个地取得单词。

❷ 管理单词部分

管理单词。最终的结果输出功能也包含在这部分。

❸ 主处理过程

统一管理以上两个模块。

对于“取得单词部分”，我们完全可以直接使用 1.3.6 节中的实现*。

get_word()中，对调用方输入的单词字符数做了限制，并且存在一些无论如何都不允许的情况*。对于这一点，其实不必过于纠结，解决方案是提供一个足够大的缓冲——1024 个字符的临时缓冲区。

对于如何定义“英语单词”，如果严密地去考虑，那就没完没了了，不妨结合现成的 get_word()中的实现——将使用 C 的宏（ctype.h）isalnum()返回真的连续的字符视作单词。

作为公开给其他部分的接口，“取得单词部分”提供了 get_word.h（参照代码清单 5-1），在需要的时候，可以通过#include 引用这个头文件。

* 好像本书的章节安排都是有预谋的哦，哈哈！

* 此时，可以考虑使用 4.2.4 节中介绍的方法。

代码清单 5-1　get_word.h

```
1:  #ifndef GET_WORD_H_INCLUDED
2:  #define GET_WORD_H_INCLUDED
3:  #include <stdio.h>
4:
5:  int get_word(char *buf, int size, FILE *stream);
6:
7:  #endif /* GET_WORD_H_INCLUDED */
```

这次的主题是“管理单词部分”。

“管理单词部分”提供了 4 个函数作为对外接口。

❶ 初始化

```
void word_initialize(void);
```

初始化“管理单词部分”。使用“管理单词部分”的一方，必须在一开始就要调用 word_initialize()。

❷ 单词的追加

```
void add_word(char *word);
```

向“管理单词部分”加入单词。

add_word()为传入的字符串动态地分配内存区域，并且将字符串保存在其中。

❸ 输出单词的出现频率

```
void dump_word(FILE *fp);
```

将利用 add_word()加入的单词按照字母顺序进行排序，并且向各单词追加它们各自出现的次数（调用 add_word()的次数），最后输出 fp 指向的流。

❹ 结束处理

```
void word_finalize(void);
```

结束“单词管理部分”。一旦结束使用“单词管理部分”，需要调用 word_finalize()。

调用 word_finalize()之后，再调用 word_initialize()，管理单词部分就回到最初的状态（清空了过去加入的单词）。

将这些整理到头文件中，如代码清单 5-2 所示。

代码清单 5-2　word_manage.h

```
 1:  #ifndef WORD_MANAGE_H_INCLUDED
 2:  #define WORD_MANAGE_H_INCLUDED
 3:  #include <stdio.h>
 4:
 5:  void word_initialize(void);
 6:  void add_word(char *word);
 7:  void dump_word(FILE *fp);
 8:  void word_finalize(void);
 9:
10:  #endif /* WORD_MANAGE_H_INCLUDED */
```

在“主处理过程”中，使用 get_word()努力地从输入流读取单词，然后通过调用 add_word()，将每次读取到的单词，不断地加入到“单词管理部分”，最后再调用 dump_word()将最终结果输出（参照代码清单 5-3）。

代码清单 5-3　main.c

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include "get_word.h"
 4:  #include "word_manage.h"
 5:
 6:  #define WORD_LEN_MAX (1024)
 7:
 8:  int main(int argc, char **argv)
 9:  {
10:      char        buf[WORD_LEN_MAX];
11:      FILE        *fp;
```

```
12:
13:     if (argc == 1) {
14:         fp = stdin;
15:     } else {
16:         fp = fopen(argv[1], "r");
17:         if (fp == NULL) {
18:             fprintf(stderr, "%s:%s can not open.\n",
                    argv[0], argv[1]);
19:             exit(1);
20:         }
21:     }
22:
23:     /* 初始化“管理单词部分” */
24:     word_initialize();
25:
26:     /* 边读取文件，边加入单词 */
27:     while (get_word(buf, WORD_LEN_MAX, fp) != EOF) {
28:         add_word(buf);
29:     }
30:     /* 输出单词的出现频率 */
31:     dump_word(stdout);
32:
33:     /* “管理单词部分”的结束处理 */
34:     word_finalize();
35:
36:     return 0;
37: }
```

代码清单 5-3 中，特意将 WORD_LEN_MAX 的值设定得大一些，但是无论如何，这只是提供给临时缓冲的空间的大小。在 add_word()内部，只是分配需要的内存区域，并且将字符串复制进去，所以不会浪费太多的内存区域。

此外，考虑到例程的简洁，本章的程序省略了对所有的 malloc()返回值检查。



补充　**头文件的写法**

写头文件的时候，必须遵守下面两个原则，

❶ 所有的头文件中，必须要有防止#include 重复的保护。
❷ 所有的头文件只#include 自己直接依赖的头文件。

“防止重复#include 的保护”是指，假设对于 word_manage.h（参照代码清单 5-2）来说，

```
#ifndef WORD_MANAGE_H_INCLUDED
#define WORD_MANAGE_H_INCLUDED
            ⋮
#endif /* WORD_MANAGE_H_INCLUDED */
```

如果像上面这么做，即使当前头文件被重复#include，其中的内容也会被无视，因此，就不会出现重复定义的错误。

下面来谈谈第 2 个规则。假设对于头文件 a.h，它依赖于头文件 b.h（使用了 b.h 定义的类型或者宏等），因此在 a.h 的开头#include 了 b.h。比如，word_manage.h 使用了结构体 FILE，所以 word_manage.h 中#include 了 stdio.h。

很多人讨厌将#include 嵌套来写*，但在 a.h 依赖 b.h 的时候，如果不采用嵌套，就需要在**所有**使用 a.h 的地方都写成下面这样，

```
#include "b.h"
#include "a.h"
```

这样做是不是有点笨拙？不光每次写得很辛苦，而且将来某个时刻 a.h 不再依赖 b.h 后，上面的两行可能就永远地留在代码中了。另外，对于这种#include 方式，开发现场的“可爱的”程序员们不可避免地会有下面的举动，

> 唉～究竟是谁依赖谁，真是完全搞不明白哦！嗯～，先把已经编译通过的.c 文件中开头的#include 部分完全复制过来吧，嘿嘿……

这样恶搞，肯定会在很多代码文件中留下大量无效的头文件引用。

啊？你问 Makefile 的依赖关系是怎么做的？**这必然是通过工具自动生成的**——这可是常识哦。

对于头文件，还有一个要点需要跟大家提前说明，那就是“应该把公有的和私有的分开”这个原则，具体的内容会在后面给大家介绍。

> 要　点
> **写头文件时必须遵守的原则：**
> **❶ 所有的头文件中，必须要有防止重复#include 的保护。**
> **❷ 所有的头文件只#include 自己直接依赖的头文件。**

* 作为 C 的编程风格的指导书，著名的《Indian Hill 编程风格手册》（参照 http://dennou-t.ms.u-tokyo.ac.jp/arch/comptech/cstyle/cstyle-ja.htm）中，也表达了“不要用#include 嵌套”这样的观点。可是，在附属于 C 处理环境的头文件中，以及广泛使用的开源软件中，随处可以看到嵌套的头文件。（第 8 次印刷注：上面的网页好像已经不存在了，但在 http://www.archive.org 中输入上面的 URL，应该可以看到以前的影子）

5.1.3　数组版

对于 word_count 的“管理单词部分”的数据结构，先让我们考虑使用数组来实现一下。

在使用数组管理单词的时候，可以考虑采取下面的方式：

❶ 将单词和其出现的次数整理到结构体中。

❷ 将这些结构体组织成数组，并且管理各单词的出现频率。

❸ 为了将单词的追加和结果输出变得更加简单，使用将数组的元素按照单词的字母顺序排序的方式进行管理。

据此写的头文件 word_manage_p.h 内容如下（参照代码清单 5-4）：

代码清单 5-4　word_manage_p.h（数组版）

```
 1:  #ifndef WORD_MANAGE_P_H_INCLUDED
 2:  #define WORD_MANAGE_P_H_INCLUDED
 3:  #include "word_manage.h"
 4:
 5:  typedef struct {
 6:      char        *name;
 7:      int         count;
 8:  } Word;
 9:
10:  #define WORD_NUM_MAX    (10000)
11:
12:  extern Word     word_array[];
13:  extern int      num_of_word;
14:
15:  #endif /* WORD_MANAGE_P_H_INCLUDED */
```

每当加入一个新的单词时，可以考虑对数组进行下面这样的操作：

❶ 从数组的初始元素开始遍历。如果发现同样的单词，将此单词的出现次数加 1。

❷ 如果没有发现相同的单词，就在行进到比当前单词“大的单词”（根据字母顺序，位置在后面的单词）的时刻，将当前单词插入到“大的单词”前面。

向数组中插入单词的方法如下（参照图 5-2）：

❶ 将插入点后方的元素顺次向后移动。

❷ 将新的元素保存在空出来的位置上。

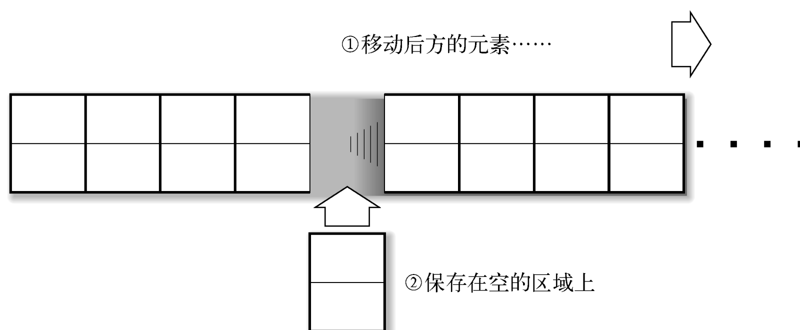


图 5-2　向数组插入元素

此时出现的问题是：**每次向数组插入元素的时候，必须要移动后方的元素**。

一般地，在初始化数组时必须要确定元素个数。当然，你可以使用 4.1.3 节中说明的方法，动态地为数组分配内存空间，此后使用 `realloc()`“嗖嗖地”将数组伸长……其实，前面曾经给大家建议过，应该避免频繁地使用 `realloc()`扩展内存区域（参照 2.6.5 节）。

如果采用下节中介绍的“链表”，就可以规避以上的问题*。

数组版的管理单词的源代码为代码清单 5-5、代码清单 5-6、代码清单 5-7 和代码清单 5-8*。

* 其实数组也不是一无是处。排序后数组的检索速度非常快，这就是数组的一个优点，关于这一点可以参照 5.1.5 节。

* 本例中没有对数组越界情况做检查。

代码清单 5-5　initialize.c（数组版）

```
 1:  #include "word_manage_p.h"
 2:
 3:  Word    word_array[WORD_NUM_MAX];
 4:  int     num_of_word;
 5:
 6:  /************************************************************
 7:   * 初始化管理单词部分
 8:   ************************************************************/
 9:  void word_initialize(void)
10:  {
11:      num_of_word = 0;
12:  }
```

代码清单 5-6　add_word.c（数组版）

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <string.h>
 4:  #include "word_manage_p.h"
 5:
 6:  /*
 7:   * 将 index 后面的元素（包括 index）依次向后方移动
 8:   */
 9:  static void shift_array(int index)
10:  {
11:      int src;    /* 被复制元素的索引 */
12:
13:      for (src = num_of_word - 1; src >= index; src--) {
14:          word_array[src+1] = word_array[src];
15:      }
16:      num_of_word++;
17:  }
18:
19:  /*
20:   * 复制字符串。
21:   * 尽管大多数处理环境都有 strdup()这样的函数，
22:   * 但是 ANSI C 规范中却没有定义这个函数，姑且先自己写一个。
23:   */
24:  static char *my_strdup(char *src)
25:  {
26:      char        *dest;
27:
28:      dest = malloc(sizeof(char) * (strlen(src) + 1));
29:      strcpy(dest, src);
30:
31:      return dest;
32:  }
33:
34:  /*******************************************************
35:   * 追加单词
36:   *******************************************************/
37:  void add_word(char *word)
38:  {
39:      int i;
40:      int result;
41:
42:      for (i = 0; i < num_of_word; i++) {
43:          result = strcmp(word_array[i].name, word);
44:          if (result >= 0)
45:              break;
46:      }
47:      if (num_of_word != 0 && result == 0) {
48:          /* 发现相同的单词 */
49:          word_array[i].count++;
50:      } else {
51:          shift_array(i);
52:          word_array[i].name = my_strdup(word);
```

```
53:             word_array[i].count = 1;
54:         }
55:     }
```

代码清单 5-7　dump_word.c（数组版）

```
 1: #include <stdio.h>
 2: #include "word_manage_p.h"
 3: 
 4: void dump_word(FILE *fp)
 5: {
 6:     int         i;
 7: 
 8:     for (i = 0; i < num_of_word; i++) {
 9:         fprintf(fp, "%-20s%5d\n",
10:                 word_array[i].name, word_array[i].count);
11:     }
12: }
```

代码清单 5-8　finalize.c（数组版）

```
 1: #include <stdlib.h>
 2: #include "word_manage_p.h"
 3: 
 4: /**********************************************************
 5:  * 管理单词部分的结束处理
 6:  **********************************************************/
 7: void word_finalize(void)
 8: {
 9:     int i;
10: 
11:     /* 释放单词部分的内存区域 */
12:     for (i = 0; i < num_of_word; i++) {
13:         free(word_array[i].name);
14:     }
15: 
16:     num_of_word = 0;
17: }
```

5.1.4　链表版

在前面一节中，我们指出了数组版中存在下面的问题：

- 中途向数组插入元素，后面的元素就必须依次向后方移动，导致效率低下*。
- 数组初始化时就需要决定元素个数。尽管可以使用 `realloc()`不断地进行空间扩展，但在数组占用了较大的内存区域的情况下，还是要尽量避免使用这种手法。

* 当然，在删除元素时，如果需要填充空出来的位置，也有必要去移动后面的元素。此时，可以使用“删除标记”这种手法，但这不能算是正当的做法，如果能不用还是不要用吧。

使用被称为**链表**的数据结构，可以避免这些问题。

链表是指将**节点**（node）对象通过指针连接成锁链状的数据结构（参照图 5-3）。

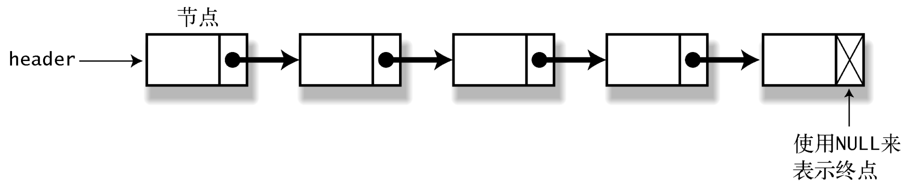


图 5-3　链表

为了通过链表来管理单词，在结构体 `Word` 中追加指向下一个元素的指针 `next`。

```
typedef struct Word_tag {
    char    *name;
    int     count;
    struct Word_tag *next;
} Word;
```

这里的 `next` 就是指向下一个元素的指针。

代码清单 5-9 就是链表版的 word_manage_p.h。

代码清单 5-9　word_manage_p.h（链表版）

```
 1:  #ifndef WORD_MANAGE_P_H_INCLUDED
 2:  #define WORD_MANAGE_P_H_INCLUDED
 3:
 4:  #include "word_manage.h"
 5:
 6:  typedef struct Word_tag {
 7:      char                *name;
 8:      int                 count;
 9:      struct Word_tag     *next;
10:  } Word;
11:
12:  extern Word *word_header;
13:
14:  #endif /* WORD_MANAGE_P_H_INCLUDED */
```

链表通过将内存单元链接起来，持续地给 `Word` 分配内存区域。和对数组通过 `realloc()`进行内存扩展不同，链表方式不需要连续的内存区域，所以不会出现效率非常低下的情况。

此外，链表元素的插入、删除都是非常方便的。对于数组，插入元素时

需要将插入点后的元素顺次向后移动，对于链表，只需简单地调整一下指针的指向就完成操作了。链表中的元素不需要在内存中连续排列。

以下是对链表的一些基本操作，

❶ 检索

通过对指针进行追踪，从链表中检索出目标元素*。

```
/* header 是链表初始元素的地址  */
for (pos = header; pos != NULL; pos = pos->next) {
    if (找到目标元素)
        break;
}
if (pos == NULL) {
    /* 没有找到目标元素后的处理 */
} else {
    /* 找到目标元素后的处理 */
}
```

* pos 是 position 的简称。

❷ 插入

在已经获得指向某个元素的指针 pos 的情况下，在当前元素的**后面**插入新的元素 new_item，下面是具体操作（参照图 5-4），

```
new_item->next = pos->next;
pos->next = new_item;
```

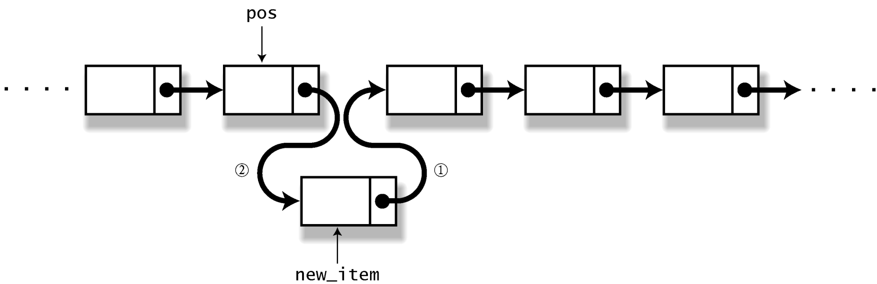


图 5-4　向链表追加元素

如果 pos 指向链表的最后一个元素，此时根据实际情况，只需要在链表的末尾追加新的元素就可以了。

这一次我们使用的是“单向链表”，因为从单向链表中的某个元素不能追溯到它前面的元素*，所以目前还不能将一个新的元素插入到已知元素的**前面**。

* 实际上，“将元素先追加在 pos 的后面，然后交换元素内部的数据内容”这样的技巧，也可以在视觉上达到将元素追加在 pos 前面的效果。在 C 语言开发中，在数据元素自身中放入指针信息构造一个链表，是很普遍的做法，如果只是移动“元素内容”，那么在确定当前元素内容被哪个指针所指向时，就比较麻烦，所以还是不要用这种方法。

❸ 删除

在已经获取指向某个元素的指针 pos 的情况下，如果需要删除这个元素

的下一个元素，可以进行下面的操作（参照图 5-5），

```
temp = pos->next;
pos->next = pos->next->next;
free(temp);
```

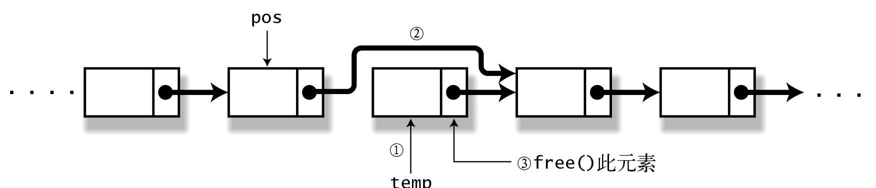


图 5-5　从链表删除元素

对于单向链表，即使是获取了某个元素的指针，也无法删除这个元素本身*。这还是因为不能追溯到它前面的元素的缘故。

* 这里也可以用“将 pos 后面的元素内部的数据内容复制到 pos，然后删除掉 pos 后面的元素”这样的技巧。但如果要删除最后一个元素，这种方法就无能为力了。

链表版的管理单词部分的源代码为代码清单 5-10、代码清单 5-11、代码清单 5-12 和代码清单 5-13。

代码清单 5-10　initialize.c（链表版）

```
 1:  #include "word_manage_p.h"
 2:
 3:  Word *word_header = NULL;
 4:
 5:  /*******************************************************
 6:   * 初始化单词管理部分
 7:   *******************************************************/
 8:  void word_initialize(void)
 9:  {
10:      word_header = NULL;
11:  }
```

代码清单 5-11　add_word.c（链表版）

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <string.h>
 4:  #include "word_manage_p.h"
 5:
 6:  /*
 7:   * 复制字符串
 8:   * 尽管大多数处理环境都有 strdup()这样的函数，
 9:   * 但是 ANSI C 规范中却没有定义这个函数，姑且先自己写一个。
10:   */
```

```
11:  static char *my_strdup(char *src)
12:  {
13:      char        *dest;
14:
15:      dest = malloc(sizeof(char) * (strlen(src) + 1));
16:      strcpy(dest, src);
17:
18:      return dest;
19:  }
20:
21:  /*
22:   * 生成新的 Word 结构体
23:   */
24:  static Word *create_word(char *name)
25:  {
26:      Word        *new_word;
27:
28:      new_word = malloc(sizeof(Word));
29:
30:      new_word->name = my_strdup(name);
31:      new_word->count = 1;
32:      new_word->next = NULL;
33:
34:      return new_word;
35:  }
36:
37:  /******************************************************
38:   * 追加单词
39:   ******************************************************/
40:  void add_word(char *word)
41:  {
42:      Word        *pos;
43:      Word        *prev;  /* 指向 pos 前一个 Word 元素的指针 */
44:      Word        *new_word;
45:      int         result;
46:
47:      prev = NULL;
48:      for (pos = word_header; pos != NULL; pos = pos->next) {
49:          result = strcmp(pos->name, word);
50:          if (result >= 0)
51:              break;
52:
53:          prev = pos;
54:      }
55:      if (word_header != NULL && result == 0) {
56:          /* 发现相同的单词 */
57:          pos->count++;
58:      } else {
59:          new_word =  create_word(word);
60:          if (prev == NULL) {
61:              /* 插入到初始位置 */
62:              new_word->next = word_header;
63:              word_header = new_word;
64:          } else {
```

```
65:                 new_word->next = pos;
66:                 prev->next = new_word;
67:             }
68:         }
69: }
```

代码清单 5-12 dump_word.c（链表版）

```
 1: #include <stdio.h>
 2: #include "word_manage_p.h"
 3: 
 4: /*****************************************************
 5:  * 将内存中的单词输出
 6:  *****************************************************/
 7: void dump_word(FILE *fp)
 8: {
 9:     Word        *pos;
10: 
11:     for (pos = word_header; pos; pos = pos->next) {
12:         fprintf(fp, "%-20s%5d\n",
13:                 pos->name, pos->count);
14:     }
15: }
```

代码清单 5-13 finalize.c

```
 1: #include <stdlib.h>
 2: #include "word_manage_p.h"
 3: 
 4: /*****************************************************
 5:  * 管理单词部分的结束处理
 6:  *****************************************************/
 7: void word_finalize(void)
 8: {
 9:     Word        *temp;
10: 
11:     /* 将所有登录的单词 free() */
12:     while (word_header != NULL) {
13:         temp = word_header;
14:         word_header = word_header->next;
15: 
16:         free(temp->name);
17:         free(temp);
18:     }
19: }
```

这里的 add_word()和数组版一样，从头开始顺序地遍历链表。行进到比当前单词“大的单词”（按照字母顺序处在后面的单词）时，将当前单词插入到这个“大的单词”前面。但是对于单向链表，在发现了比当前单词“大的单词”的时候，无法在其前面插入当前单词。因此，在例程中，声明了一个

称为 prev 的指针，它指向 pos 前面一个元素。

word_finalize()通过 free()释放链表中所有的元素。用一句话来表述就是：从链表的初始元素开始将元素一个个剥离，然后 free()掉。

此时，很多人也许会写出下面的代码，

```
Word *pos;

/* 从链表的初始位置开始顺序地遍历，然后进行free()（意图上）*/
for (pos = word_header; pos != NULL; pos = pos->next) {
    free(pos->name)
    free(pos);
}
```

以上这段代码是**错误的**。将 pos 通过 free()释放掉之后，应该是不能引用 pos->next 了吧。但是这段程序在一般处理环境中**还是能很好地执行**。但在某些环境中也会给我们带来麻烦（参照 2.6.4 节）。



补充 **头文件的公有和私有**

本章在统计单词出现频率的应用中，对于“管理单词部分”，介绍了“数组版”和“链表版”两个版本的程序。

可是，“管理单词部分”向外部公开的头文件 word_manage.h 却没有发生任何改动。因此，即使将“管理单词部分”的实现方式从数组变成链表，对于使用它的一方（main.c）也没有必要做任何的修改，也不需要再重新编译（只需重新连接一次就可以了）。

“管理单词部分”向外部公开的的头文件 word_manage.h 和用于“管理单词部分”内部共享数据的头文件 word_manage_p.h，是完全分离的。因此，无论“管理单词部分”内部的实现怎样变化，都不会给“使用的一方”带来影响。

我一般将向外部公开的头文件称为“公共头文件”，将用于内部共享数据的头文件称为“私有头文件”。

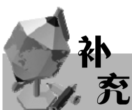
私有头文件的内部大多都使用了公共头文件提供的类型、宏或者函数，所以私有头文件在大部分的情况下，都#include 了公共头文件。

但是，无论是间接地还是直接地，公共头文件都**不可能#include** 私有头文件。这就好比对于一个公司来说，将向公司外部发布的宣传资料同时在公司内部公开，是没有问题的，但是公司内部的商业资料（内部机密）要是流传到公司外部，那就……!!

如果遵守以上的规则，私有头文件中记述的内容就不会暴露给使用它的模块，因此，在大型程序的开发中，可以使用多团队来各自分担开发任务。

此外，在开发大型程序的工程，对于函数名、全局变量名，公共头文件中声明的类型名、宏名，会使用“命名规约”来防止名称冲突*。但是这次的例程中，我们还没有做到这么规范。

* 对于 Java、Ada 或者 C++这样的自身有命名空间控制的语言，编程时是不需要依赖命令规约的。但是在现实中，java.awt.List 和 java.util.List 还是经常会发生冲突，让人感到困惑。现在，在 Swing 中，也在类名加上了“J”这样的前缀，这就说明命名规约还是必要的。



补充　怎样才能同时使用多份数据

在如今的应用程序中（MS-Word、MS-Excel 等），同时打开多个文件，并且分别在不同的窗口进行编辑，已经是很正常的事情了。

可是，在这次的“管理单词部分”，无论是数组版还是链表版，数据的“源头”都保存在全局变量中。因为全局变量“**在同一时刻只存在一份**”，所以无法同时使用多份数据。

如果要解决这个问题，不妨将保存数据的“源头”的部分定义成结构体，对于链表版，可以定义成下面这样，

```
typedef struct {
    Word *word_header;
} WordManager;
```

然后，在 word_initialize()中，使用 malloc()给 WordManager 分配新的内存区域，并且返回指向这片内存区域的指针。

```
WordManager *word_initialize(void);
```

另外，对于“管理单词部分”的其他函数，都将指向 WordManager 的指针作为第 1 个参数进行传递。

```
void add_word(WordManager* word_manager, char *word);
void dump_word(WordManager* word_manager, FILE *fp);
void word_finalize(WordManager* word_manager);
```

通过这样的方式，使用“管理单词部分”的一方能够管理多个 WordManager，当然也就可以同时使用多份数据。

此外，对于在公共头文件中声明的 WordManager 类型中，Word 类型也是必须的，因此，此时对使用方屏蔽了“使用了链表”这个实现上的细节。

其实对于使用方来说，只有指向 WordManager 的指针才是必须的，并没有必要知道内部的细节。这种情况下，只声明一个“**不完全类型**”倒是一种常用手法。

```
typedef struct WordManager_tag WordManager;

然后在私有头文件中，将实体填充到 struct WordManager_tag 中，

struct WordManager_tag {
    Word *word_header;
};
```

5.1.5 追加检索功能

目前的 word_count 仅仅是读取文本文件，然后输出统计数据。既然好不容易统计出文本文件中各单词的出现频率，如果程序还能提供“这个单词出现了几次”这样的功能是不是更好？

因此，在“管理单词部分”中追加下面的接口，

```
int get_word_count(char *word);
```
返回通过 word 指定的单词的出现次数。

最简单的方法就是，对保存在数组或者链表中的单词，**从头开始顺序地遍历**，这样肯定能检索到目标单词。这种方法称为**线性检索**（linear search）。但是对于数组中的数据已被排序的情况，可是考虑使用效率更高的**二分检索**（binary search）。

以下是二分检索的过程，

❶ 定位到数组中央的元素。
❷ 如果此元素是
  ① 要检索的目标元素，结束检索。
  ② 比要检索的目标元素小，就对数组中此元素后面的元素进行相同操作。
  ③ 比要检索的目标元素大，就对数组中此元素前面的元素进行相同操作。

其实我们在查字典的时候，也经常使用这个方法。

代码清单 5-14 是“数组版”的 get_word_count()的实现例程。

代码清单 5-14　get_word_count.c

```
 1:  #include <stdio.h>
 2:  #include <string.h>
 3:  #include "word_manage_p.h"
 4:  /*****************************************************
 5:   * 返回某单词出现的次数
 6:   *****************************************************/
 7:  int get_word_count(char *word)
 8:  {
 9:      int left = 0;
10:      int right = num_of_word - 1;
11:      int mid;
12:      int result;
13:
14:      while (left <= right) {
15:          mid = (left + right) / 2;
16:          result = strcmp(word_array[mid].name, word);
17:          if (result < 0) {
18:              left = mid + 1;
19:          } else if (result > 0) {
20:              right = mid - 1;
21:          } else {
22:              return word_array[mid].count;
23:          }
24:      }
25:      return 0;
26:  }
```

这种方法只对数组有效。对于链表，因为很难（迅速地）找到“中央的元素”，所以不能使用二分检索。

任何一种数据结构都不是万能的，各自都有自己的优缺点，我们必须根据实际需要选择合适的数据结构。比如在元素的个体相对比较大的情况下，就推荐使用指针的可变长数组（参照图 5-6）。

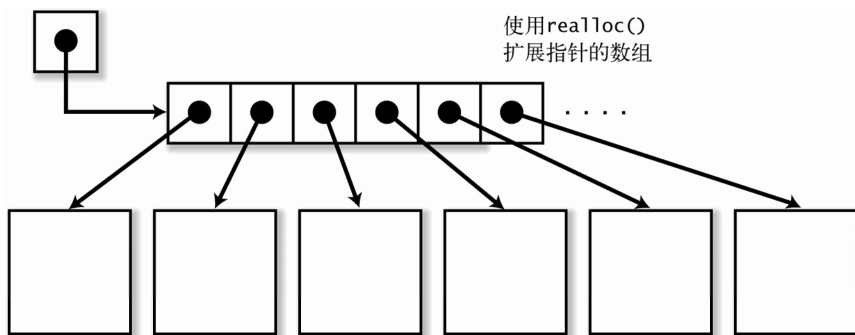


图 5-6　使用指向元素的指针的数组

如果是这种方式，即使每一个元素都比较大，指针的数组自身占用的内存区域也不会变得很大，因此也许可以使用 realloc()来不断地增加数组的内存空间*。这种情况下，可以使用二分法做检索操作。

* 确实，在元素个数很多的情况下，使用 realloc()来扩展数组的内存区域，有点……



题外话

倍倍游戏①

在数据量少的情况下，无论是使用线性检索还是二分检索，执行效率上并不会产生显著的差距。可是，随着数据量的增加，二分检索在执行效率上就会凸显出**压倒性的**优势。

对于上面的结论，也许有人会产生疑问：

> 真的吗？二分检索的逻辑复杂性肯定也会影响执行效率吧？是不是会和自身原本的优势相互抵消呢？结果就是，二分检索和线性检索的效率不会相差很大呢！

对于线性检索，随着数据量的增加，检索时间按比例延长。但是，使用二分检索的时候，数据每增加一倍，检索次数只会增加一次。因此，即使是数据量大到现实中无法想象的地步，也能够使用很短的时间将数据检索出来。

为了更感性地理解这一点，下面给大家打一个比方。

假设手头有一张报纸，它的厚度是 0.1mm，将它对折，厚度就变成了 0.2mm，再对折，就变成了 0.4mm 了吧。

那么，这样折了 100 次后，厚度变成了多少呢？

当然，也许我们折不了这么多次，此时切成两份再堆积起来，也能达到同样的目的。还是让我们单纯从数字上来回答：对折 100 次后的厚度是多少？

1 米左右？NO～NO～答案是，约 **134 亿光年**。如果你不信，你可以用手头的计算器算算（1 光年为 9 兆 4600 亿 KM）。*

也就是说，如果使用二分检索，对于“134 亿光年/0.1mm”条数据，仅仅 100 次循环就完成检索了。这么大的数据量，如果选择使用线性检索……嗯，恐怕在我有生之年是完成不了这个工作了。

* 所以对于我来说，在机器猫的道具中，比起“地球毁灭炸弹”（参照瓢虫系列漫画第 7 卷），“加倍药水”（参照天道虫连环漫画第 17 卷）更让我感到“坑爹”，哈哈。

① 来自曾吕利新左卫门和丰臣秀吉之间的一段对话，曾吕利新左卫门对丰臣秀吉说：“作为奖赏，第 1 天给 1 粒米，次日给第 1 天的加倍的量，再次日给第 2 天加倍的量，如此这般。”——译者注

5.1.6　其他的数据结构

这一节介绍一下其他的数据结构。

▲双向链表

之前为大家介绍过“单向链表”。

因为“单向链表”不能“回溯”，所以它有以下缺点，

❶ 在向链表中追加元素的时候，必须要知道追加点**前面的元素**。
❷ 从链表中删除元素的时候，必须要知道被删除元素**前面的元素**。
❸ 很难逆向遍历链表。

如果是**双向链表**（double linked list），就可以解决上面的问题。

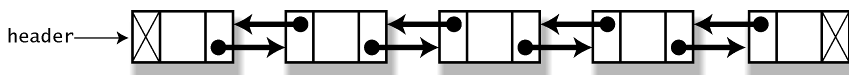


图 5-7　双向链表

使用结构体定义链表，大概就像下面这样：

```
typedef struct Node_tag {
    /* Node 自身的数据 */

    struct Node_tag *prev; /* 指向前一个元素的指针 */
    struct Node_tag *next; /* 指向后一个元素的指针 */
} Node;
```

可是，双向链表也有不足的地方，

❶ 因为每个元素都需要 2 个指针，所以会消耗多余的内存。
❷ 指针操作过于频繁， 编程容易出现 bug。

▲树

UNIX 和 DOS 的“目录”，以及 Windows 和 Macintosh 的“文件夹”都是层次结构。这样的数据结构好似一棵倒置的树，所以我们称之为“**树**”（tree）结构。

树中的各元素称为**节点**（node）。最根部的节点称为**根**（root）。某节点 A 在节点 B 的直接下方的位置，A 就称为 B 的**子**（child）节点，B 就称为 A 的**父**（parent）节点。比如，在图 5-8 中，Node5 为 Node2 的子节点，Node2 为 Node5 的父节点。父节点和子节点之间的连接线称为**枝**（branch）。

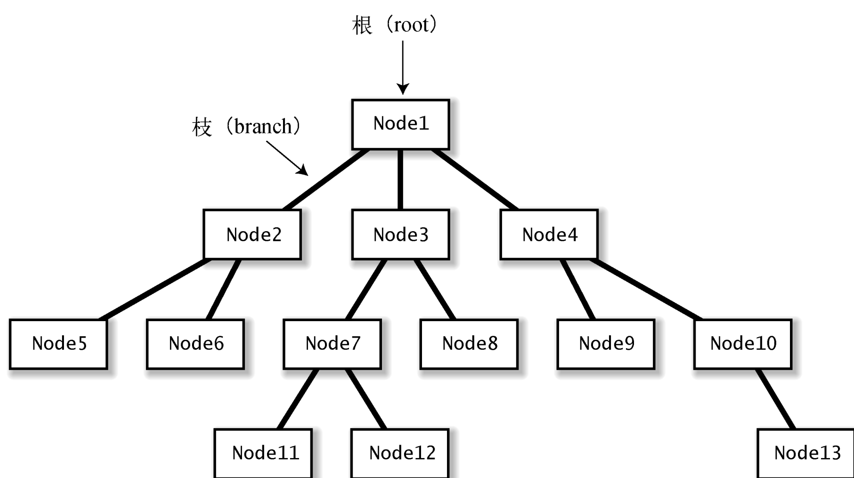


图 5-8　树

使用 C 语言表示树的时候，一般定义一个像下面这样的结构体（参照图 5-9），

```
typedef struct Node_tag {

    /* 节点自身的数据 */

    int              nchildren;  /* 子的个数 */
    struct Node_tag **child;     /* 此指针指向的是通过 malloc()分配的内
                                    存区域，其中包含指向子节点的指针的可
                                    变长数组*/
} Node;
```

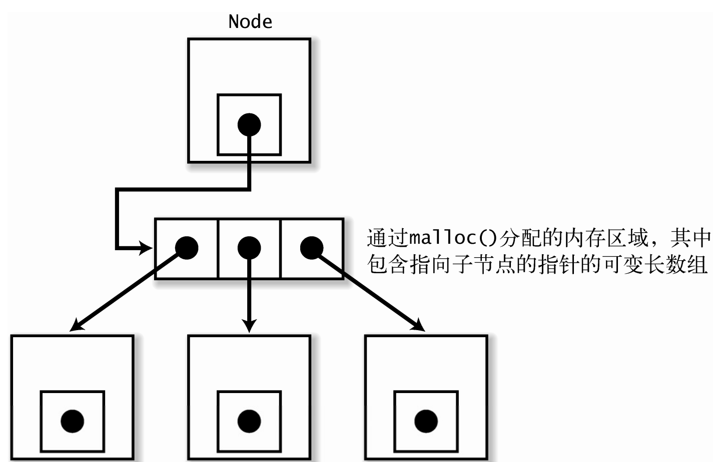


图 5-9　如果使用 C 来表示树……

如果树中的任意节点最多只有两个子节点，此时称这种树为**二叉树**（binary tree）*。

* 问题：最多只有一个子节点的树，怎么称呼？——是“链表”哦。

使用二叉树的**二叉检索树**（binary search tree）这样的数据结构，也适用于前面的 word_count 这个案例。

二叉检索树是所有节点满足以下条件的二叉树（参照图 5-10），

❶ 节点 p 左侧的子都小于 p
❷ 节点 p 右侧的子都大于 p

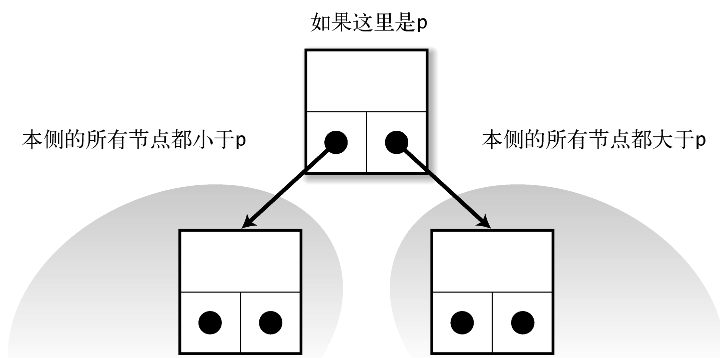


图 5-10　二叉检索树

对二叉树元素的追加和检索，按以下方式进行。

❶ **追加**

从根开始顺序地追加，如果追加的元素小于当前节点，就向左移动，大于当前节点，就向右移动。如果遇到相等的节点或者 NULL，就将元素追加到当前位置。

❷ **检索**

从根开始顺序地遍历，如果追加的元素小于当前节点，就向左移动，大于当前节点，就向右移动。一旦查找到目标节点，检索结束。如果遇到 NULL 的元素，就说明树中不存在目标元素。

如果二叉树的结构比较理想，追加和检索应该还是非常快速的。可是，最坏的情况下（比如，对 word_count 按照字母排序进行处理），二叉树只不过是个链表而已。

在不同情况下，单纯的二叉树在效率上的表现是参差不齐的，而且还容易引起 worst-case*，所以现实中，单纯的二叉树可能并不实用。有时候可以用 B 树、AVL 树等来代替单纯的二叉树。

* 输入的数据从一开始就有序的情况也很多见。

▲哈希

日常生活中，我们怎样管理大量的数据（假设它们记录在卡片上）呢？而且对这些数据的追加、删除和检索等操作还十分频繁。

干事严谨的人，会将记录数据的卡片排好顺序进行保存。此时，使用二分法检索应该是个不错的选择。可是，在需要保证数据有序的情况下，插入卡片是相当花费时间的（尤其当卡片保存在多个抽屉中时）。

懒散的人会把所有的卡片一齐扔到一个箱子中，想找一张卡片的时候，他也许会从头开始一张张地去寻找。这种方式对于追加操作很简单，但是检索过程却格外地花费时间。

如果是严谨而又不失灵活的人，会考虑将卡片放进几个分好类的箱子中进行管理。

所谓**哈希**（hash）就是基于上面第三种想法进行构造的。

典型的哈希结构“链式哈希*”就是通过在**哈希表**（hash table）中保存链表的方式，实现对元素的管理（参照图 5-11）。

* 除此之外，还有完全哈希、线性再哈希、非线性再哈希等种类的哈希结构。但是我很少见到它们用于现实中。

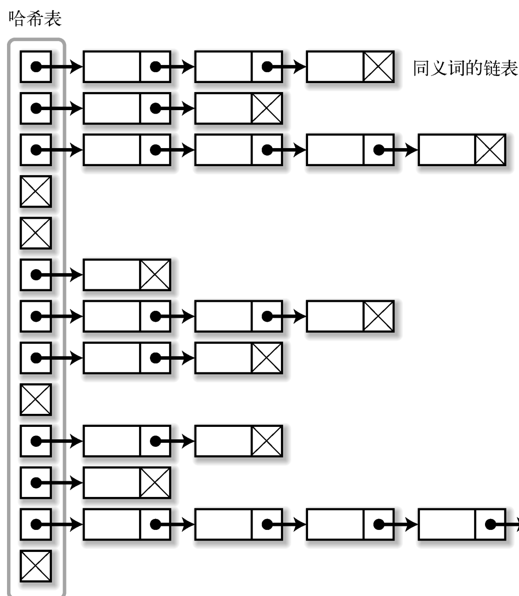


图 5-11　链式哈希

此时，通过**哈希函数**来决定保存某元素的哈希表的下标。哈希函数根据检索的 key（对于 `word_count` 的 `get_word_count()`，key 就是单词的字符串），会尽量返回一个无规律的值。在将字符串作为 key 的情况下，哈希函数通常会使用“将每个字符进行移位运算，然后进行加法运算，其结果除以哈希表的元素个数，最后得到余数”这样的算法。如果不走运，哈希函数对于不同的 key 也会返回相同的值，此时这些 key 被称为**同义词**（synonym）。

在检索的时候，以检索 key 求得哈希表的下标，从与此下标关联的链表中检索元素。如果哈希函数尽量返回均匀的值，和哈希表相连的链表就会变短，自然就会大幅地提高检索的速度。

在编译器的标识符管理和 AWK、Perl 等语言的**哈希数组***等实现中，经常会用到哈希表*。

* 不光可以使用整数，还可以用字符串作为下标的数组。

* Perl 从 Ver.5 开始，将“哈希数组”称为“哈希”……虽说哈希数组可能是用哈希的方式实现的，但是有意地将内部“实现手段”体现在名称这些表面的东西上，对于我来说，有点难以理解。

5.2　案例学习 2：绘图工具的数据结构

5.2.1　案例的需求

这次让我们考虑开发一个绘图工具，来进一步实践。假定程序的名称为“X-Draw”。

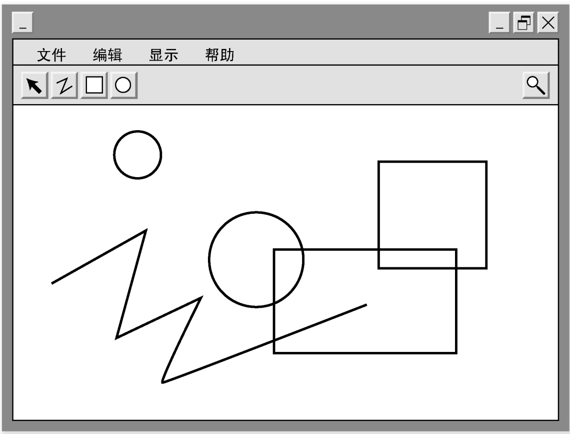


图 5-12　绘图工具“X-Draw”的界面

在 X-Draw 中，可以绘制下面的图形*，

* 也许让大家失望了，这只是个很简陋的绘图工具。

- ❑ 折线（有任意个顶点的折线）
- ❑ 长方形（不考虑相对坐标轴倾斜的长方形）
- ❑ 圆形（不考虑弧形和椭圆形）

考虑到篇幅，我没有在书中提供所有的程序代码。不管怎么说，窗体程序还是非常依赖运行环境的。

这里只会跟大家说明实现 X-Draw“数据结构”的“头文件”部分。

5.2.2 实现各种图形的数据模型

首先让我们考虑一下怎么表示折线、长方形、圆形等图形。

在第 4 章已经跟大家讨论过折线的表示方式，在这里我们依然采用这种方式。

```
typedef struct {
    double      x;
    double      y;
} Point;

typedef struct {
    int         npoints;
    Point       *point;
} Polyline;
```

对于长方形，可以使用处于对角线上的两个点来表示：

```
typedef struct {
    Point       minPoint; /* 左下的坐标 */
    Point       maxPoint; /* 右上的坐标 */
} Rectangle;
```

对于圆形，可以用圆心和半径来表示：

```
typedef struct {
    Point       center; /* 圆心 */
    double      radius; /* 半径 */
} Circle;
```



补充　坐标系的话题

读到这里，可能有很多人有下面的疑问，

嗯？坐标值怎么都搞成了 `double`？绘制图形的画布不是用像素来表示的吗？我记得坐标值应该是 `int` 啊？

确实，无论是 X Window System 的图形库 Xlib，还是 Windows 的 API，还是 Java 的 AWT，坐标值都是以像素为单位的整数*。

尽管如此，程序内部还是不应该使用以像素为单位的整数值来表示坐标。如果使用以像素为单位的整数值，首先需要解决的问题就是如何实现图像的放大表示。很多工具软件只能做到 200%、400%这样的整倍地放大，使用起来似乎不是很方便。此外，对于稍稍复杂的图形，将它们组合，然后通过操作顶点再将其一会儿放大，一会儿缩小，图形就变得粗糙不堪了吧。

计算机上的图形大多都是用像素（这种方式并不是很细腻）来表示的，这是由显示设备**自身的条件**造成的，并不是用户一开始就想使用像素来绘制和表示图形*。

因此，正确的解决方案应该是，首先定义逻辑上假定的**用户坐标系***，在图形显示的时机再转换成**设备坐标系**（参照图 5-13）。

因为有“像素”这样的源于设备条件的限制，用户坐标系的坐标值使用 double*。另外，设备坐标系的原点大多在左上，而用户坐标系是遵循数学坐标将原点设置在左下，我认为这种做法比较通俗易懂*。

做一些乘法运算和减法运算，可以简单地将用户坐标和设备坐标进行转换*。

此外，Windows 从设备坐标系独立出来一种自定义的逻辑坐标系，它可以以毫米为单位进行图形绘制。但……此时坐标值的类型却为 `short int`，这样的设计思路，好像完全超出了我的理解能力……

* Windows 的情况稍微有点复杂，关于这一点，后面会有说明。

* 在画笔工具的情况下，也可能不是这样。

* 有时也称为世界坐标系或者逻辑坐标系。

* 也可以使用 `float`。在 C 中，浮点数类型大多都会变成 `double`，在内存充裕的情况下，最好还是使用 `double`。

* 这还是要根据用途来决定，在文字处理软件中绘图的时候，因为文字是横着从左上写到右下的，如果考虑到这一点，将原点设定在左上也许更方便。

* 最一般的做法，可以采用矩阵。

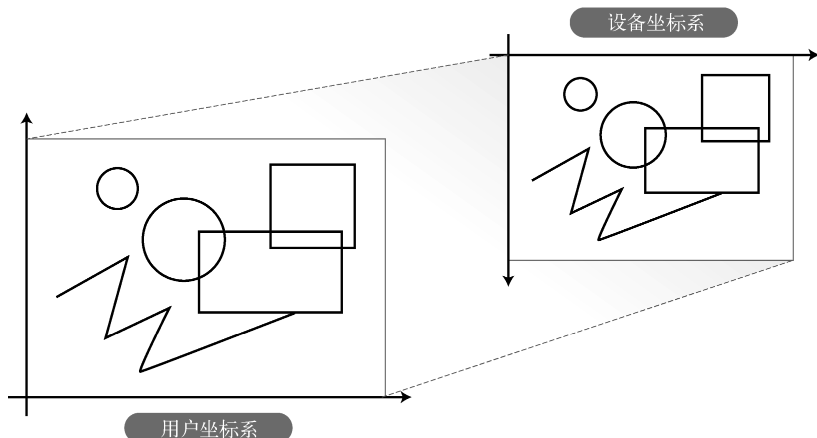


图 5-13　坐标系变换

5.2.3 Shape 型

5.2.2 节中讨论了各图形的表示方法。对于绘图工具，还需要管理大量各种各样的图形进行管理。

在这里，我们使用 Shape 这样一个结构体来表示一个“图形”。

对于“图形”来说，它应该有颜色这样的属性。如果内部被涂满，那就还有填充方式这样的属性。下面使用枚举的方式来表示这些属性。

```
typedef enum {
    COLOR_BLACK,    /* 黑 */
    COLOR_BLUE,     /* 蓝 */
    COLOR_RED,      /* 红 */
    COLOR_MAGENTA,  /* 品红 */
    COLOR_GREEN,    /* 绿 */
    COLOR_CYAN,     /* 青*/
    COLOR_YELLOW,   /* 黄*/
    COLOR_WHITE     /* 白 */
} Color;

typedef enum {
    FILL_NONE,      /* 不填充 */
    FILL_SOLID,     /* 实心填充 */
    FILL_HATCH,     /* 斜线图案填充 */

    FILL_CROSSHATCH /* 交叉图案填充 */
} FillPattern;
```

在一开始我们是无法预测图形的数量的，所以考虑使用 malloc()为 Shape 动态地分配内存区域，然后通过链表对它们进行管理。

考虑到以下原因，我们选择双向链表，

- 对所有的图形进行重绘的时候，“从后面的图形开始顺序地”绘制，能够以正确的顺序表示各图形。
- 使用按下鼠标的方式选择图形的时候，需要从“前面的图形开始顺序地”检查哪个图形才是选择对象*。

* 在绘图工具中，基于通过鼠标来选择图形（pick），经常需要运用几何方法来计算距离。

对于 Shape，它可能代表折线，可能代表长方形，也可能代表圆，使用 C 的结构体和共用体来表示 Shape 是惯用的手法（下面就是实际的例子）。

根据上面的分析，可以用下面的方式来表示 Shape 类型：

```
typedef enum {
    POLYLINE_SHAPE,
    RECTANGLE_SHAPE,
```

```
    CIRCLE_SHAPE
} ShapeType;

typedef struct Shape_tag {
    /* 画笔（轮廓）的颜色 */
    Color       pen_color;
    /* 填充样式，FILL_NONE 的时候不填充 */
    FillPattern fill_pattern;
    /* 填充颜色 */
    Color       fill_color;
    /* 图形的种类 */
    ShapeType   type;
    union {
        Polyline        polyline;
        Rectangle       rectangle;
        Circle          circle;
    } u;
    struct Shape_tag *prev;
    struct Shape_tag *next;
} Shape;
```

ShapeType 是一个用于区别 Shape 种类的枚举类型，根据 Shape 的成员 type，可以判定 Shape 的种类。

假设，将指向 Shape 的链表的头指针放到 head 这样一个变量中，“绘制所有图形的程序”大致可以写成下面这样：

```
Shape *pos;

for (pos = head; pos != NULL; pos = pos->next) {
    switch (pos->type) {
      case POLYLINE_SHAPE:
        /* 调用绘制折线的函数 */
        draw_polyline(pos);
        break;
      case RECTANGLE_SHAPE:
        /* 调用绘制长方形的函数 */
        draw_rectangle(pos);
        break;
      case CIRCLE_SHAPE:
        /* 调用绘制圆形的函数 */
        draw_circle(pos);
        break;
      default:
        assert(0);
    }
}
```

因为是从链表的第一个元素开始绘制图形，所以链表后面的元素会被表示在前面。

在大多的绘图工具中，通过点击鼠标选择一个图形，然后可以将它移动

到最前面或最后面。此时，需要将图形对应的元素在链表中进行移动，图形要移动到最前面，元素就要移动到链表的末尾；图形要移动到最后面，元素就要移动到链表的最前面。

做这样的元素移动，那是链表的强项。

5.2.4 讨论——还有别的方法吗

到这里为止我们讨论过的方法好像都还不错，但还是让我们再想想有没有更好的解决方案。

可能有人会有下面的想法，

> 俺所知道的绘图工具中，经常需要的功能倒不是什么绘制折线，绘制“线段”倒挺常见的。虽然菜单中也有折线，但是经常使用的是“线段”。因此，是不是应该也有一个 Line 这样的类型呢？

我平时使用 UNIX 的 tgif 这个工具比较多一些，在这个工具的菜单中前面出现的就不是“线段”，而是“折线”。确实，在 Windows 和 Macintosh 中的绘图工具的菜单中出现的大多还是“线段”。

“线段”经常出现在菜单的最前面，证明对于用户来说，相比“折线”，绘制“线段”功能的使用会更加频繁*。但即使是这样，如果用户选择了位于菜单最前面的“线段”选项，程序能够很简单地绘制出“顶点数目为 2 的折线”，那么就不需要特地再搞出一个什么 Line 类型了吧。

* 到底是不是这样，我还真说不好。但至少作为软件设计者，肯定应该这样去考虑。

即使 Windows 中的绘图工具总是将折线功能藏在菜单层次结构的里面，但这种功能毕竟是存在的，并且，作为折线的修正功能，肯定也会有“增减折线顶点的功能”吧。此外，恐怕也会有增加线段的顶点的功能吧。因此，线段的内部使用折线来实现，应该是一个不错的方案。如果定义了新的 Line 类型，就又需要开发一个“一旦在 Line 上添加了顶点，就要将它转换成 Polyline 类型”这样的功能了。

这样说来，是不是有人又会有下面的想法：

> 嗯，长方形可以用顶点数为 5 的折线来表示，那 Rectangle 类型是不是也可以不要了？

这可不行！假设我们选中一个顶点进行拖拽，对于长方形和折线图形如何变化，我们的期待肯定是不一样的吧？

让我们再把目光投向内部实现，此时也许又会出现下面的观点：

> 折线必须要实现“增减顶点”的功能，所以对于折线的 Point，不应该用可变长数组，而应该使用链表来表示。
>
> 追加顶点的时候，往往是在顶点和顶点之间插入新顶点，因此，相比使用数组，链表的插入操作更简单。

这倒是一个比较麻烦的问题，不能断然说哪个观点就是正确的。

但是，Point 类型只包含两个 double 元素，相对来说比较小。对于绘图工具，折线顶点的数量也是可以预期的。因此，Point 的数组也不会变得很大，可以说不会发生导致执行效率低下的现象（追加顶点时，使用 realloc() 不断地分配内存区域，或者在插入顶点时，整体移动后方的元素）。倒是为了使用链表，在成员中追加指针，或者为每一个 Point 使用 malloc()分配管理空间这样的行为反而会引起严重的问题。

> 如果对 malloc()的管理区域比较在意的话，可以使用可变长结构体来构造 Polyline 类型呀！

对啊，如果使用 4.3.1 节中介绍的技巧，就可以不用调用 malloc()为 Point 数组来分配新的内存区域了！……但是，由于 Polyline 不是 Shape 结构体的最后成员，一旦使用这个技巧，就会破坏掉 prev 和 next 的数据。

调整一下 Shape 结构体成员的顺序看上去好像可行，但折线的顶点个数一旦发生变化，就必须为每一个 Shape 调用 realloc()，当前的 Shape 的地址也会随之发生变化（有可能）。由于 Shape 在双向链表中，它被前后两个 Shape 所指向，一旦移动了地址，自然麻烦也随之而来。因此，这个方法也不靠谱。

> 前面我们使用了共用体。共用体会根据其最大的一个成员来分配内存空间，这是不是有点浪费内存？

倒是有点道理。但是这次的 Polyline、Rectangle 和 Circle 的大小，在大多数处理环境中都是差不多的。

如果这些类型的大小完全不同，当然也不能无视内存的浪费。此时，也可以考虑使用“指针共用体”*，然后通过 malloc()给 Polyline、Rectangle 和 Circle 分配内存区域。

* 当然也可以使用 void*，但此时完全不知道指针指向什么类型，所以从代码的可读性上考虑，还是应该使用指针的共用体。

```
typedef struct Shape_tag {
        ⋮
```

```
    union {
        Polyline        *polyline;  ┐
        Rectangle       *rectangle; ├ 指针的共用体
        Circle          *circle;    ┘
    } u;
    struct Shape_tag *prev;
    struct Shape_tag *next;
} Shape;
```

此时，刚才的“应该把 Polyline 构造成可变长结构体”这种方案也就具备了现实的意义。

但是，正如刚才描述的那样，因为 Polyline、Rectangle、Circle 的大小并没有很大差异，考虑到使用 malloc()需要额外的管理内存，或者消耗在调用 free()上的精力，所以我不会考虑通过别的方式分配内存。

> 在 Shape 中放进 prev 和 next 的做法也让人觉得不太自然。Shape 只是一个“图形”，使用链表来管理还是使用数组来管理，这是使用者的自由，为什么我们一开始就**先入为主地认为 Shape 天生就应该通过双向链表来管理**呢？这一点本身就很奇怪。

这个批评不无道理。我们经常将链表和本来无关的 Shape 放在一起使用，此时有人会想：那就将 prev 和 next 设定为 NULL 好了。但将原本就不需要的对象作为成员，你不觉得奇怪吗？

因此，让我们来看看下面这个方案。这个方案不需要在 Shape 中放入 prev 和 next，而是像下面这样定义一个 LinkableShape 类型，Shape 作为这个类型的一个成员（参照图 5-14）。

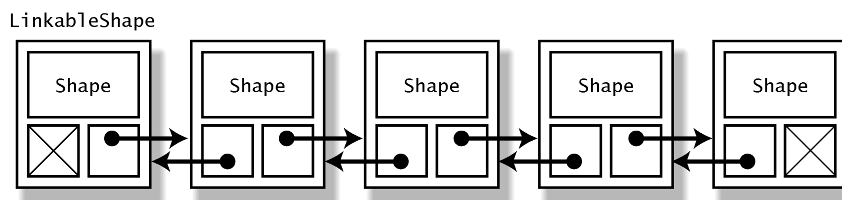


图 5-14　Shape 的持有方式（其 2）

```
typedef struct LinkableShape_tag {
      Shape shape;
      struct LinkableShape_tag *prev;
      struct LinkableShape_tag *next;
  } LinkableShape;
```

或者写成这样，

```
typedef struct LinkableShape_tag {
      Shape *shape;
      struct LinkableShape_tag *prev;
      struct LinkableShape_tag *next;
  } LinkableShape;
```

在图 5-14 的实现中，保存 Shape 到链表的时候，需要将 Shape 整个复制到链表中。复制这样的操作肯定会改变 Shape 的地址，后果就是丢失了指向 Shape 的指针。

如果采用图 5-15 的方法，就没有这样的担心了。但会出现另外一个问题：随着 malloc()调用次数的增加，会浪费一些管理区域的内存，而且也要在 free()调用上花费相当的精力。

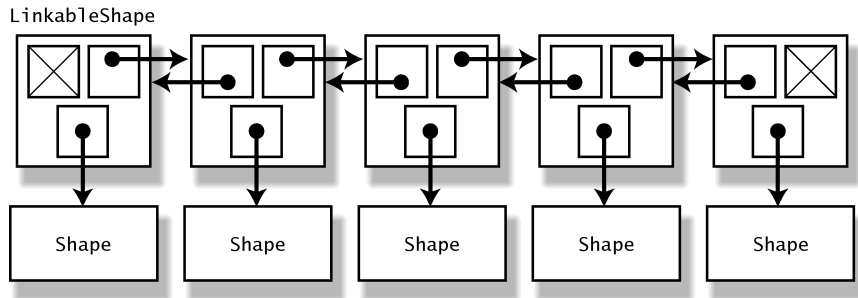


图 5-15　Shape 的持有方式（其 3）

但是，对于本例这样的结构体，考虑到实现简单，在 Shape 中放入 prev 和 next 其实也没有什么问题。

通过上面种种分析，我们应该能体会到，对于数据结构的设计，其实**说到底就是权衡利弊**的过程，根本就不存在所谓的“银弹”[①]。对于数据的特征以及使用方式进行细致的分析，并且选择最合适的手法，这是程序设计者的责任。同时，设计者还要熟知 malloc()、realloc()的内部实现机制。

另外，设计者不但需要根据现状决定使用方法，还要考虑到将来的扩展性，以及执行速度、内存使用这两个方面的最优化设计。最后，还要考虑到编程的可行性。在设计数据结构这一点上，最能体现设计者的水平。

① 银弹：出自《人月神话》，指那些可以解决所有问题的方法论，可以理解成“一招鲜”。——译者注



补充　什么都放的链表

进一步考虑图 5-15 的方案，应该还有以下的方法：

```
typedef struct Linkable_tag {
    void  *object;
    struct Linkable_tag *prev;
    struct Linkable_tag *next;
} Linkable;
```

Linkable的成员变量object的类型是void*,所以它可以指向任意类型。也就是说，在 Linkable 类型不依赖于 Shape 类型，它可以存储任意类型。

大部分情况下，双向链表都会拥有指向初始元素和末尾元素的指针，为了能够持有整个双向链表，有必要定义下面这样的类型：

```
/* 持有全体双向链表的类型 */
typedef struct {
    Linkable *head;  /* 初始的元素 */
    Linkable *tail;  /* 末尾的元素 */
} LinkedList;
```

双向链表是使用很频繁的一种数据结构，但使用双向链表进行开发的时候，每次都重复做“在某元素前面插入一个元素”这样无聊的编程，既浪费时间又容易引起 bug。

此时，可以使用 LinkedList 或者 Linkable 这样的类型，如果将双向链表的常用操作整理成函数库来加以利用,就可以省去那些无聊的重复编程。

可是，在大型项目中，这个方法也有致命的弱点。造成这个弱点的起因就是“什么都可以扔进去”。向保存 Shape 的链表中无限制地放入“白萝卜”还是“胡萝卜”总是会出问题的（编译时不会出错）。

此外,也完全不知道 LinkedList 中究竟保存了什么。假设,使用 Point 的链表来表示 Polyline：

```
typedef struct Point_tag {
    double x;
    double y;
    struct Point_tag *prev;
    struct Point_tag *next;
} Point;

typedef struct {
    Point *head;
    Point *tail;
} Polyline;
```

看一眼上面的代码，基本可以知道数据结构中的内容。

```
    typedef struct {
        LinkedList list;
    } Polyline;
```

如果这样，那就真不知道 list 是什么东东了。就算好不容易搞明白它是个链表，你也无法知道它其实是个“Point 类型的链表”（除非你加上注释）。这对程序的可读性和可维护性产生了恶劣的影响。

顺便介绍一下，Java 的集合类库中也有和此处介绍的 void*方式类似的实现*。

* java.lang.Object 相当于 C 的 void*，它们都是最邪恶的指针。

对于 C 这样的骨灰级语言，已经没有必要解决这样的问题了。在一些最新的语言中，为了持续维持 LinkedList 的通用性和杜绝 void*的危险性，实现了叫做“泛型（genericity）”和“模板（template）”的功能。

Java 中缺少这么一个重要的功能，真是让人不解*。

* C#里也似没有……[1]

对于 C++、Ada 这样可以操作对象实体的语言，泛型伴随着复制执行代码的问题。但是，对 Java 这样只能使用指针的语言来说，只要在编译时加入一些检查就可以简单地实现这个功能*。

* 现在，也出现嵌入泛型功能的 Java 编译器。（参照 http:// www.cs.bell-labs.com/who/wadler/gj）。（第 8 印注：在 JDK1.5 中被正式采用。）

5.2.5　图形的组合

几乎在所有的绘图工具中，都可以将几个图形进行组合，然后当作一个图形去操作。本节中也考虑将这个功能在 X-Draw 中实现。

首先定义一个 Group 类型。虽然 Group 类型需要包含很多 Shape，但是 Shape 自身可以实现双向链表，所以只要在 Group 中定义指向初始元素和最后元素的指针就可以了。

```
    typedef struct {
        Shape   *head;
        Shape   *tail;
    } Group;
```

所谓图形组合就是“将几个图形组合成一个图形”的功能。被组合到一起的图形组，其自身也可以说是一种“图形”。因此，可以考虑将“Group”加入到枚举类型 ShapeType 中。

```
    typedef enum {
        POLYLINE_SHAPE,
        RECTANGLE_SHAPE,
        CIRCLE_SHAPE,
        GROUP_SHAPE
    } ShapeType;
```

① C#从.NET2.0 开始也具备了泛型和模板的功能。——译者注

但是，可能有人觉得这么做有些不和谐，让 Group 和 Polyline、Rectangle 相提并论真的有些奇怪。

因为 Group 也是一个 Shape，所以我倒认为这么做未尝不可。只是对于 Shape 来说，它虽然有自己的颜色或者填充图案，但“组合后的图形”的颜色就不是固定的。将图形进行组合，然后去变更其颜色，整组的图形颜色都会发生变化。但之后如果解除了图形的组合，其每一个图形都不会变回原来的颜色，因此你不能说这是一个“变更组合图形的颜色”的功能，准确地应该说这是一个“变更组内所有图形的颜色”的功能。

在这里，让我们首先把 Shape 分类成折线、长方形这样的“基本图形”和“组”。

```
typedef enum {
    PRIMITIVE_SHAPE,
    GROUP_SHAPE
} ShapeType;

typedef struct Shape_tag {
    ShapeType           type;
    union {
        Primitive       primitive;
        Group           group;
    } u;
    struct Shape_tag *prev;
    struct Shape_tag *next;
} Shape;
```

然后，将到现在为止 Shape 中持有的数据放到 Primitive 类型中。

```
typedef enum {
    POLYLINE_PRIMITIVE,
    RECTANGLE_PRIMITIVE,
    CIRCLE_PRIMITIVE
} PrimitiveType;

typedef struct {
    /* 画笔（轮廓）的颜色 */
    Color       pen_color;
    /* 填充样式，FILL_NONE 的时候不填充 */
    FillPattern fill_pattern;
    /* 填充颜色 */
    Color       fill_color;
    /* 图形的种类 */
    PrimitiveType   type;
```

```
        union {
            Polyline        polyline;
            Rectangle       rectangle;
            Circle          circle;
        } u;
    } Primitive;
```

此时，包含“组”的 Shape 数据结构，就变成图 5-16 的形式。看上去好像和 5.1.6 节中列举的例子有些不同，其实它也是一种树结构。

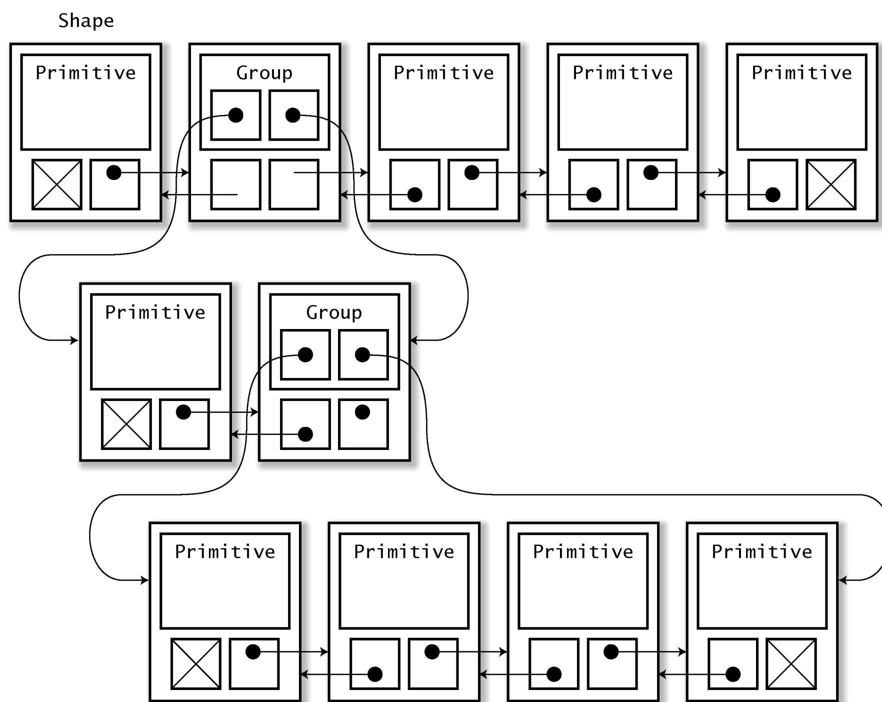


图 5-16　包含“组”的 Shape 数据结构

包含到此为止涉及的所有类型的头文件，如代码清单 5-15 所示，

代码清单 5-15　Shape.h

```
1:  #ifndef SHAPE_H_INCLUDED
2:  #define SHAPE_H_INCLUDED
3:
4:  typedef enum {
5:      COLOR_BLACK,                /* 黑 */
6:      COLOR_BLUE,                 /* 蓝 */
7:      COLOR_RED,                  /* 红 */
```

```
 8:      COLOR_MAGENTA,              /* 品红 */
 9:      COLOR_GREEN,                /* 绿 */
10:      COLOR_CYAN,                 /* 青 */
11:      COLOR_YELLOW,               /* 黄 */
12:      COLOR_WHITE                 /* 白 */
13:  } Color;
14:
15:  typedef enum {
16:      FILL_NONE,                  /* 不填充 */
17:      FILL_SOLID,                 /* 实心填充 */
18:      FILL_HATCH,                 /* 填充斜线图案 */
19:      FILL_CROSSHATCH             /* 填充交叉阴影线 */
20:  } FillPattern;
21:
22:  typedef enum {
23:      POLYLINE_PRIMITIVE,
24:      RECTANGLE_PRIMITIVE,
25:      CIRCLE_PRIMITIVE
26:  } PrimitiveType;
27:
28:  typedef struct {
29:      double      x;
30:      double      y;
31:  } Point;
32:
33:  typedef struct {
34:      int         npoints;
35:      Point       *point;
36:  } Polyline;
37:
38:  typedef struct {
39:      Point       minPoint;       /* 左下的坐标 */
40:      Point       maxPoint;       /* 右上的坐标 */
41:  } Rectangle;
42:
43:  typedef struct {
44:      Point       center;         /* 圆心 */
45:      double      radius;         /* 半径 */
46:  } Circle;
47:
48:  typedef struct {
49:      /* 画笔（轮廓）的颜色 */
50:      Color       pen_color;
51:      /* 填充样式，FILL_NONE 的时候不填充 */
52:      FillPattern fill_pattern;
53:      /* 填充的颜色 */
54:      Color       fill_color;
55:      /* 图形的种类 */
56:      PrimitiveType   type;
57:      union {
58:          Polyline        polyline;
59:          Rectangle       rectangle;
60:          Circle          circle;
61:      } u;
```

```
62:  } Primitive;
63:
64:  typedef struct Shape_tag Shape;
65:
66:  typedef struct {
67:      Shape   *head;
68:      Shape   *tail;
69:  } Group;
70:
71:  typedef enum {
72:      PRIMITIVE_SHAPE,
73:      GROUP_SHAPE
74:  } ShapeType;
75:
76:  struct Shape_tag {
77:      ShapeType           type;
78:      union {
79:          Primitive       primitive;
80:          Group           group;
81:      } u;
82:      struct Shape_tag *prev;
83:      struct Shape_tag *next;
84:  };
85:
86:  #endif /* SHAPE_H_INCLUDED */
```

Shape 类型的定义依赖于 Group 类型，Group 类型的定义又依赖于指向 Shape 类型的指针，因此它们之间形成了相互依赖的关系。第 64 行利用不完全类型声明了 Shape 类型。76 行以后给 struct Shape_tag 定义了实际的内容。

至于使用此数据结构的程序，比如“绘制所有图形”的程序，如代码清单 5-16 所示。

代码清单 5-16　draw_shape.c

```
 1:  #include <stdio.h>
 2:  #include <assert.h>
 3:  #include "shape.h"
 4:
 5:  void draw_polyline(Shape *shape);
 6:  void draw_rectangle(Shape *shape);
 7:  void draw_circle(Shape *shape);
 8:
 9:  void draw_primitive(Shape *shape)
10:  {
11:      switch (shape->u.primitive.type) {
12:        case POLYLINE_PRIMITIVE:
13:          /* 调用绘制折线的函数 */
14:          draw_polyline(shape);
```

```
15:         break;
16:       case RECTANGLE_PRIMITIVE:
17:         /* 调用绘制长方形的函数 */
18:         draw_rectangle(shape);
19:         break;
20:       case CIRCLE_PRIMITIVE:
21:         /* 调用绘制圆的函数 */
22:         draw_polyline(shape);
23:         break;
24:       default:
25:         assert(0);
26:     }
27: }
28:
29: void draw_all_shapes(Shape *head)
30: {
31:     Shape       *pos;
32:
33:     for (pos = head; pos != NULL; pos = pos->next) {
34:         switch (pos->type) {
35:           case PRIMITIVE_SHAPE:
36:             draw_primitive(pos);
37:             break;
38:           case GROUP_SHAPE:
39:             draw_all_shapes(pos->u.group.head);
40:             break;

41:           default:
42:             assert(0);
43:         }
44:     }
45: }
```

图形的具体绘制方式依赖于各窗口图形系统，在代码清单 5-16 的第 5 ~ 7 行，假定了传入指向 Shape 类型的指针就可以绘制对应图形的几个函数*。

* 从常识来说，非 static 函数的原型声明不应该写在.c 文件中。外部函数的原型声明必定写在头文件中，被多个头文件共用。此处的例程，只是为了通过编译（不出警告），暂时做了这样的原型声明。

第 39 行通过对 draw_all_shapes()递归调用来实现图形组合的绘制。在这种需要遍历树结构的情况下，使用递归调用是常用的手法。

5.2.6　继承和多态之道

本书是一本关于 C 语言的书籍，所以到现在为止，只是在 C 语言的功能范围内讨论了绘图工具的数据结构。

其实，前面介绍的方法中存在比较严重的问题，它就是：

> 为了区别处理各种图形，switch case 这样的语句散落在程序的各个角落。

代码清单 5-16 的第 11 行开始出现的 switch case，不只是出现在图形绘制的时候，在使用鼠标选择图形而进行距离计算的时候，以及将整体图形保存在文件中，或者从文件中加载图形的时候，都会出现 switch case 的身影。

像这样在程序中到处写 switch case 的编程风格，一旦增加了一种图形，就必须在分散在各处的 switch case 中挨个追加 case。不但麻烦，还很容易疏漏*。

* 在代码清单 5-16 中的 default 中的 assert()，可以在早期就发现这种疏漏。

在 C++或者 Java 等面向对象的语言中，通过使用继承和多态，可以在很多实现中回避 switch case。

简单地说，面向对象的语言有以下这些功能：

❶ 在面向对象的类（粗暴地说，它类似于结构体）中，不但可以有变量，还可以放入函数，这些函数我们称为“方法”。

❷ 在面向对象的语言中，可以对现有的类进行扩展，做出一个新的类，这种行为被称为继承（inheritance）。比如，Polyline 继承了 Shape。

❸ 继承类的方法可以覆盖（override）被继承类的方法。

使用这个功能，可以将 draw()这个方法放到 Shape 中，然后针对 Polyline 或者 Rectangle 做 draw()的具体实现，“绘制所有图形的程序”就可以大致写成下面这样（C++风格）：

```
for (pos = head; pos != NULL; pos = pos->next) {
    pos->draw();    ←─调用 pos 指向的 Shape 的 draw()方法
}
```

这么写的话，如果 pos 是 Polyline 就自动调用 Polyline 的 draw()方法，如果 pos 是 Rectangle 就自动调用 Rectangle 的 draw()方法。就没有必要再使用 switch case 来区分处理了。这样的特征我们称为多态（polymorphism）。

如果想要用 C 来实现多态，可以考虑让每一个 Shape 持有一个指向结构体的指针，而在结构体中包含函数指针。**这样的做法很流氓**。其实，在 X Window System 的 GUI 工具包 Xt Intrinsics 和 GTK+中，就是通过这样的方法勉强地实现了多态。但这种方式毕竟还是太暴力了，在一般开发中还是建议大家老老实实地使用 switch case。

编程语言是在不断发展的，每一种新的语言都会推出一些有用的新功能。



补充 真的可以将 draw()放到 Shape 中吗？

将 draw()方法定义到 Shape 中，然后利用多态的特征将处理分开，很多面向对象的入门书都将这个案例作为例题来讲解。比如，在《编程语言 C++ 第 3 版[9]》中，就利用 Shape 的例子来说明多态。

在我看来，对于最多几万行代码的小规模程序，“在 Shape 中定义 draw()”这种手法还是很有效的。可是，如果是像 CAD 这样庞大的系统，有时就需要尽量回避这种手法。

定义了 Shape 等类型的 shape.h，是全体程序中最主要的头文件，其中的大量内容被所有的程序引用。因此，必须非常谨慎地定义 shape.h 的内容，因为一旦内容确定了，以后再修改就不是那么简单的了。

因为 draw()方法的实现特别依赖于窗口的图形系统，如果将 draw()方法放到 shape.h 中，就会发生严重的可移植性的问题。像 C++那样将方法的声明和实现分开还好，如果是 Java 那就比较麻烦了①。

> 正是这种情况下我们才要考虑**设计模式**[10]呀！如果想要不依存窗口图形系统，为什么不考虑使用 Bridge 模式呢？

可能有人会有上面的想法。可是，抽象出可以对应现实中所有窗口图形系统的绘图接口，是非常困难的。比如在如今的图形系统（X Window、Windows、Java AWT）中，“将线从这里画到那里”这样的即时绘图，和很久以前作为标准的 GKS*就有概念上的差异。

* Graphical Kernel System，它是 ISO 制定的二维图形库的国际标准。GKS 通过“segment”采用了显示列表模型。

因此，在根本概念发生变化的时候，对于程序这方面，废弃掉原来的程序，重新开发也许就能解决问题。但是，相比程序，系统数据的生命周期要长很多，所以需要尽可能不改变原来的数据结构（这里就是指 shape.h）。

另外，在 CAD 等系统使用的数据中，图形（形状）数据不只是被 draw()所用。比如，利用 CAD 设计出的图形会被保存在文件中，其他的应用程序也可以读取这些文件中的数据，并且可以对它们进行解析。对于解析的处理来说，shape.h 也许是不可缺少的，但是其中的 draw()其实是完全用不上的。“数据”往往会超越原本设计其“数据结构”的人的意志，被很多其他的应用程序使用。

① Java 中根本没有头文件，在这里，原著作者先把话题指向头文件，然后拿 C++和 Java 进行对比，这是不公平的。况且 Java 中也可以通过 interface 将声明和实现分开。在此书翻译过程中，译者认为原著作者对 Java 有着很深的偏见和误解。——译者注

那么，对于 C 语言，最终的方案是什么？将数据和处理过程分开是一个好的解决方案。问题是，为了区分图形种类而产生的 switch case。比起数据的生命周期，试图废弃一部分程序逻辑时，也不是只能使用 switch case。除此之外，还可以考虑使用 Vistor 模式*。在 Vistor 模式中，Polyline 和 Rectangle 并存于 Vistor 一侧，所以这种手法和 switch case 相比好像换汤不换药。或者在所有的 Shape 中，只定义将“形状”变换成折线的方法，而将 draw()方法放到类的外面；抑或，在 Shape 中只保留一个指向“定义了运行时执行逻辑的对象”的指针，然后在从文件中加载 Shape 实例时，通过 Abstract Factory 模式实例化出特定的运行时对象……

唉，总是有无尽的烦恼纠缠在程序设计者周围。

* 这也是一种设计模式。

5.2.7　对指针的恐惧

大多的绘图工具，都有图形“复制”功能。比如，如果是 Windows 上的工具，通过选择图形→复制→粘贴，就可以完成图形的复制。

```
/* 将指向被复制图形的指针放在 new_shape 中 */
Shape *new_shape;

new_shape = malloc(sizeof(Shape));
*new_shape = *shape;     ←—通过结构体赋值将 Shape 结构体赋予 new_shape

/* 将 new_shape 连接到链表的末尾 */
```

同学，你知道上面的代码中，哪里有问题吗？

如果被复制的图形是长方形或者圆，上面的方法是没有缺陷的。可是，一旦使用上面的方法对折线进行复制，会怎样呢？

Polyline 中只是持有指向它的坐标群（Point 的可变长数组）的指针，坐标群自身其实是保存在其他的内存区域中的。因此，即使通过结构体一次赋值的方式对 Shape 进行复制，坐标群自身却并没有被复制。结果就是，多个 Shape 共享了同一个坐标群。

这种状态显然不是我们原本期待的结果。只要变化了一个折线的坐标，其他折线的坐标也会跟着一起变化。还有，在删除某折线时使用了 free()释放坐标群的内存区域，就会发生重大的问题。其中的缘由已经在 2.6.4 节中向大家说明过了。

就像这样，如果指针指向了一个偏离程序员意图的对象，调试是异常困难的。上面的折线的例子还算比较简单，一旦图形中更加复杂的数据结构的

指针纠结到一起，结局往往惨不忍睹。

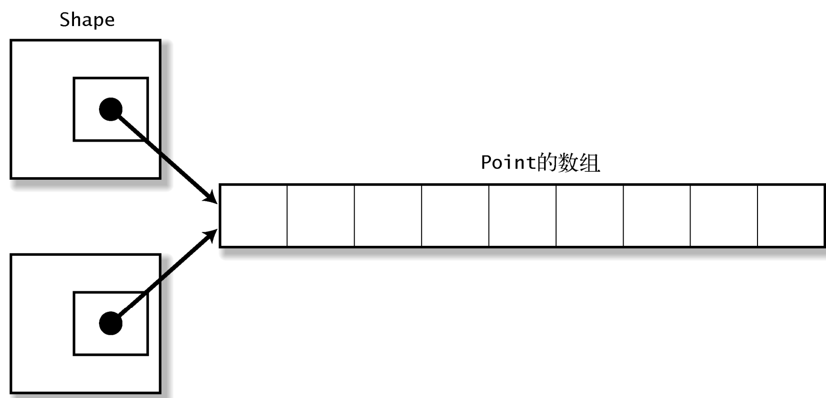


图 5-17 多个 Shape 共享一个坐标群

人们有时说："指针很可怕。"他们大多是想表达下面的意思吧，

> C 语言的指针一旦指向一个奇怪的地址，程序往往会在一个不起眼的地方崩溃。

其实，我认为真正的可怕之处在于：

> 因为引用关系的相互纠葛而导致程序调试非常困难。

前者的恐惧是 C 语言的特性带来的，而后者的恐惧，伴随着所有的持有指针的语言*。

随便提一下，Pascal 的作者 Niklaus Wirth 认为，数据结构中的指针对应于用于逻辑跳转的 goto（《算法+数据结构=程序》[11] p.192）①。确实，对此我也深有同感，但至少在目前，不使用指针，还是无法构造出复杂的数据结构*。

* 有 GC（Garbage Collection）功能的语言，也只能回避 free() 的相关问题。

* 正如一度提倡使用 if、for、while 这样的控制结构来取缔 goto 一样，如果普及使用集合类库，程序员不用顾及指针，也可以轻松操作像链表这样频繁使用的数据结构。尽管如此，我并不认为完全不使用指针的情况下，能解决所有的问题。

5.2.8 说到底，指针究竟是什么

本章从"数据结构"这个侧面，对指针进行了说明。

在单向链表、双向链表、树、哈希（链式）的图解中，"箭头"必然会登场。这个"箭头"，就是指针。

在设计数据结构的时候，如果不使用"箭头"，就很难设计出"像样的"

① goto 是目前很多编程规范中提倡尽量避免使用的逻辑跳转方式。在此，Niklaus Wirth 的意思是指针和 goto 一样，应该尽量避免使用。——译者注

数据结构。因此，“指针”是构造“像样的”数据结构时必需的概念，如今“像样的”程序都会使用指针。

如果要说起 C 语言指针特有的功能，那就不得不提到它和数组之间微妙的可交换性。关于这一点，在第 4 章进行了完整的说明。

对于 C 语言，我们经常听到下面的观点：

C 语言和硬件关系密切，为什么这么说，因为 C 语言中有指针。

对于早期的编程，或者现在有时也能见到的操作系统内核编程，也许可以认为指针和硬件关系密切。

但只要你能很好地理解第 4 章的那些常用用法，还有第 5 章中介绍的将指针作为箭头来使用的方法，仅仅是使用 C 语言来开发应用程序应该是游刃有余了。

我想，“将指针作为箭头来使用”这个“真正的指针的使用方法”，对于其他语言也同样适用。

第6章 其他——拾遗

6.1 陷阱

6.1.1 关于 strncpy( )

C语言使用空字符来结束字符串。

printf()和strcpy()都把这一点作为默认的前提条件。此外，编译器也会在字符串常量的末尾自动加上空字符。

可是，strncpy()却令人迷惑地打破了这个规则。

strncpy()像strncpy(dest, src, len);这样使用，从src复制最大长度为len的字符到dest中。当src的长度大于len时，dest不会以空字符结束*。

* 反过来，如果src比len短，就会使用空字符补足剩余的长度。这个规则也是有点奇怪的。

因此，冒冒失失地使用了strncpy()，之后再使用printf()、sprintf()、strcpy()等处理dest，由于其末尾可能没有空字符，进而可能会发生处理越界，以至于破坏大片内存区域的数据。**因此，strncpy()是危险的。**

> 要　点
> **如果使用strncpy()，请注意它可能会产生没有空字符结尾的字符串。**

但也有完全相反的观点：

> 对于 strcpy()，在 src 过长的情况下，很容易破坏内存区域。而 strncpy()可以通过指定 len 来阻止内存的被破坏。所以，strncpy()更安全。

可是，即使是通过对 len 的设定“当场”阻止了内存破坏的发生，但作为结果，还是会产生没有空字符结尾的奇怪的字符串，依然会发生由于使用 sprintf()而导致的内存破坏。倒不如说，“bug 最终爆发的现场远离制造 bug 的地方”这种结果，在性质上更为恶劣。

对于“只是复制 len 长度的字符，即使之后的字符被切掉也没问题”这种需求，是不是应该考虑写一个 1.2.5 节补充内容中介绍的 my_strncpy()这样的函数呢？依我之见，strncpy()应该用来实现 my_strncpy()这样的函数，或者用于对大型机中常见的定长字段的操作。

下面我也来扯个闲篇儿，某日某地点，我听到两个人的对话：

> 有个哥们儿写了这么一行代码，
> strncpy(dest, src, strlen(src) + 1);
> “这，这是啥啊，明明可以用 strcpy()”
> “说是用 strncpy()更安全……”

这真不是我自创的噱头，这可是真事儿！

6.1.2 如果在早期的 C 中使用 float 类型的参数

如今已经很少有人使用 ANSI C 以前的 C 了，但我认为还是不能绕过“早期的 C”来理解函数参数的类型。

函数的原型声明是从 ANSI C 以后开始导入的，之前的 C 中，函数的声明像

```
double sin();
```

这样只能指定返回值，而不能指定参数。

其实，上面的三角函数 sin()的参数应该是 double。那么，在调用 sin()的时候，如果传入 float，会导致什么样的结果呢？

在 ANSI C 的 math.h 中，sin()声明如下：

```
double sin(double x);
```

因此，即使向形参中传递 float 类型，编译器也会自动将其转换成 double 类型。

那么，对于 ANSI C 以前的 C，究竟会发生什么呢？答案就是——float 类型的参数**还是会被转换成 double**。

ANSI C 以前的 C，会将表达式中的 float 类型依次转换成 double。假设，我们需要做一次 float 类型的加法运算，然后将结果保存在 float 类型中，其内部过程如下，

❶ 将两边的变量转换成 double
❷ 进行 double 类型的加法运算
❸ 将结果变换成 float 类型

因此，**与直接使用 double 类型相比，肯定是 float 类型的加法运算速度比较慢**，于是在不知不觉中流传起来“不要使用 float 类型”这样的说法*。

* 当然，在构造较大的数组的时候，大部分的情况下还是使用 float 比较节约内存。

对于函数的参数会发生同样的问题。所以在使用 sin()的时候，应该没有必要去分辨参数是 float 还是 double。这样看上去挺方便，但对于本来就是“将 float 作为参数的函数”，又会发生什么问题呢？

在我现在使用的编译器（gcc）中，可以通过开关选项（-traditional）让编译器按照 ANSI C 以前的 C 进行处理。此外，还可以通过-S 选项输出汇编代码。通过这两个选项，尝试编译下面的两段代码（参照代码清单 6-1 和代码清单 6-2）。

代码清单 6-1　float.c

```
1:  void sub_func();
2:
3:  void func(f)
4:  float   f;
5:  {
6:      sub_func(&f);
7:  }
```

代码清单 6-2　double.c

```
1:  void sub_func();
2:
3:  void func(d)
4:  double  d;
5:  {
6:      sub_func(&d);
7:  }
```

＊ 如果再说明得细致一些，其实作为辅助信息的文件的名称是不同的。

对输出的汇编代码进行比较后，**得到了完全相同的输出结果**＊①。也就是说，在形参类型为 float 的情况下，ANSI C 以前的 C 的编译器会一声不吭地将参数解释成 double（挺可气的）。

而且在上面的例子中，向 sub_func()中传递的是指向形参的指针。对于代码清单 6-1 的 sub_func()来说，肯定应该接受“指向 float”的指针吧。但是，f 却被擅自地解释成了 double，这种传递结果自然是不正确的。

顺便说一下，对于整型也会发生同样的事情。在 C 中，比 int 小的整型如果出现在表达式中，同样会被依次地转换成 int。但是，在整形的情况下，一旦参数被作为 int 接受，就会缩小为原来的类型，所以不会发生上面的问题。

```
/*
 * 对于 void func(short s)这个函数定义，
 * 编译器会生成和下面的 C 语言代码等同的机器代码
 */
void func(int s_temp)
{
    short s = s_temp;
          ⋮
}
```

请大致浏览一下标准库函数。在 math.h 数学运算库中，清一色地都使用了 double。在 stdio.h 和 ctype.h 中，有很多操作字符的函数（如 putchar()）的参数也声明成了 int。

ANSI C 以前的 C 原本不能传递 float 类型，虽然针对 char 和 short 做了补救措施，但由于加入了多余的处理而导致了执行效率低下。

至少对于 ANSI C 以前的 C 来说，遇到整型基本作为 int，遇到浮点型基本作为 double 来考虑。

6.1.3　printf()和 scanf()

对于 ANSI C 以前的 C，比 int 小的类型会被依次地转换成 int，float 会被依次地转换成 double，所以不会出现接受 float 类型参数的函数；针对 char 和 short，编译器通过别的方式做了补救措施。

因为 ANSI C 有原型声明，所以无论是 char 还是 float，都可以直接传递。

可是，对于 printf()这样的具有可变长参数的函数，原型声明对可变长

① 根据不同的环境，输出结果可能会有一些差异。——译者注

部分的参数是不产生任何影响的。因此，这部分的参数同样会被编译器进行类型转换操作。也就是说，比 int 小的类型会被转换成 int，float 会被升级成 double。

结果就是，不能向 printf()传递 char 类型和 float 类型。

> 啥？对于 printf()，不是有%f 用于表示 float，%lf 用于表示 double 吗？

其实，这不能不说是个误解（恐怕是来源于 scanf()的转换修饰符）。对于 printf()，float 和 double 共用了%f。在 printf()中使用%lf，其实这种行为根本没有在规范中定义（如果提高 gcc 的警告级别，编译器会对%lf 的使用提出警告）。

同样，char 和 short 也可以用%d 来表示。

反过来，在带有可变长参数的函数一侧，

```
va_arg(ap, char)
va_arg(ap, short)
va_arg(ap, float)
```

这样的写法也是经常发生的错误。

此外，scanf()使用了和 printf()非常相似的转换修饰符。对于那些在使用 printf()时已经习惯将 float 和 double 都用%f 表示的程序员，对 scanf()往往有着同样的期待。

可是，因为向 scanf()传递的是指针，所以并没有放入类型信息。因此，如果想在 scanf()中使用 double，必须指定%lf。

6.1.4 原型声明的光和影

最近，ANSI C 以前的 C 确实已经开始慢慢绝迹了。以前经常会发生“将老的 C 转换成 ANSI C”的工作[①]。

ANSI C 是像下面这样进行函数定义的：

```
int func(int hoge, int piyo)
{
```

① 看到这里，干对日软件外包的同学笑了。对日软件外包中，这种活是特别多的，一般把这种活叫做“移行”。——译者注

```
        ⋮
}
```

ANSI C 以前的 C，却是写成下面这样：

```
int func(hoge, piyo)
int hoge;
int piyo;
{
        ⋮
}
```

因为 ANSI 同时也允许老的写法，所以即使不修改成新的写法，也能通过编译。但是，从 ANSI C 开始导入的函数原型声明，对发现程序员的编程错误是非常有益的。所以，建议大家尽量使用新的函数定义方法。

将老的函数定义一个一个改过来，的确是一件比较繁琐的事。

那就不要去修改函数定义本身，只要在头文件中加上原型声明不就完事儿了嘛！

应该有人是这样想的吧？我也这么想过。

但是，请你再回想一下之前说明过的内容。

在没有函数原型声明的情况下，比 `int` 小的参数的类型会被依次地转换成 int，而 float 类型的参数会被依次地转换成 `double` [这种转换称为**默认实参提升**（default argument promotion）]。因此，在接受参数的一侧生成的代码总是处理“转换后的相对较大的数据类型”。

反过来，在提供函数原型声明的情况下，因为参数以原来的类型传递，所以在接受参数的一侧，总是生成针对原来的数据类型的机器代码。

可是，函数的定义和函数的调用可能存在于完全不同的编译单元中。那么，在编译函数定义的时候，编译器究竟根据什么来判断应该生成哪种机器代码呢？——根据函数的定义的新旧形式*。

* 几乎所有的处理环境都是这样的。对于调用没有原型定义的函数，规范完全没有规定究竟应该怎么做。

对于旧的函数定义，一旦调用提供原型声明的函数，在函数定义的一侧期待的就是经过“默认实参扩展”后的数据类型，但实际中如果传递没有经过扩展的数据类型，就可能会发生问题。

为了防患于未然，应该在定义函数的文件中`#include` 声明函数自身原型的头文件。如果这么做了，但凡是正经的编译器，都会给出原型和函数定义

不一致这样的警告。无论在什么情况下，只要想让编译器帮我们找出函数原型和函数定义不一致的地方，就必须在函数定义的代码中#include 函数原型*。如果原型和函数定义不一致，那么好不容易在 ANSI C 中引入的机制简直就成了摆设了（甚至可以说是有害的）。

* 实际上，以前也曾经遇到过对函数定义和原型之间的参数一致性不做检查的编译器。

> 要 点
> **在函数定义的代码文件中，必须#include 包含此函数自身原型声明的头文件。**

一旦决定了使用原型声明，调用该函数所有代码文件也**必须#include** 包含原型声明的头文件。但人总是会犯错误的，所以还是建议提高编译器的警告级别，一旦调用没有原型声明的函数就让编译器向我们给出警告。

此外，为了提高执行速度，有些编译器往往不使用栈，而是使用寄存器来传递参数。但对于可变长参数的函数，这些编译器还是使用了栈来传递参数，但至于“是不是可变长参数的函数”，调用方在原型声明以外的地方是无法判断的。因此，在这样的处理环境中，如果不#include stdio.h，printf()是不能顺利执行的。

> 要 点
> **如果调用提供了原型声明的函数，就必须#include 原型声明。**

6.2 惯用句法

6.2.1 结构体声明

对于这种习惯用法，可能会有不同的观点。但是我在声明结构体的时候，必定会同时给出 typedef 定义。

```
typedef struct {
    int a;
    int b;
} Hoge;
```

此外，在没有特别需要的情况下，我一般不写 tag*。在需要写 tag 的时候，一般像下面这样，在定义的类型名称后面追加_tag。

* 这是由于，在特别需要的情况下写 tag，tag 的存在表明了“正在被前方引用”这层意思。也有人主张“因为不知道什么时候需要，所以必须写 tag”这样的观点。

```
typedef struct Hoge_tag {
    int a;
    int b;
} Hoge;
```

还有，对于结构体、共用体、枚举的 tag 名，因为持有和一般的标识符不同的命名空间，所以它们也可以写成下面这样：

```
typedef struct Hoge {
    int a;
    int b;
} Hoge;
```

因为写成这样则与 C++意思相同，所以也有很多人喜欢这种写法。

在声明结构体的时候，虽然可以同时定义这个结构体类型的变量，但我不会这么做。一想到 typedef，就完全没有了这么写的冲动，因为类型的声明和变量的定义根本就是不同的概念，所以我认为还是分开书写比较好。

```
/* 我不会这样写 */
struct Hoge_tag {
    int a;
    int b;
} hoge;    ←—— 声明struct Hoge_tag 类型的变量
```

随便再提一下，虽然声明结构体的成员时，可以和声明一般的变量一样，一次性声明多个变量，但是我不会这么做。

```
/* 我不会这样写 */
typedef struct {
    int a, b;
} Hoge;
```

6.2.2　自引用型结构体

为了构造链表和树，我们会声明包含指向相同类型的指针的结构体。

这样的结构体，好像一般都称为“自引用型结构体”——为什么是“好像”？C 语言的入门书籍中倒是这么称呼的，但是在开发现场，我从没听到过有人使用这个称呼。称之为“自引用型结构体”，其实没有任何特殊的理由。

但是，对于这种结构体的声明，还是有必须要留意的地方。

```
typedef struct Hoge_tag {
    int                 a;
    int                 b;
    struct Hoge_tag     *next;
} Hoge;
```

在这种情况下，在声明成员 next 的时候，typedef 还没有结束，类型 Hoge 还不能被使用，所以 next 还只能声明成 struct Hoge_tag*。

或者也可以写成下面这样：

```
typedef struct Hoge_tag Hoge;

struct Hoge_tag {
    int     a;
    int     b;
    Hoge    *next;
};
```

6.2.3 结构体的相互引用

在 3.2.10 节中我们说过，对于相互引用的结构体，应该像下面这样只是将 tag 提前声明：

在下面的代码中，Man 持有指向“妻”的指针，Woman 持有指向“夫”的指针。

```
typedef struct Woman_tag Woman;    ←提前对 tag 进行类型定义

typedef struct {
        ⋮
    Woman *wife;  /* 妻 */
        ⋮
} Man;

struct Woman_tag {
        ⋮
    Man *husband; /* 夫 */
        ⋮
};
```

以前，一提出上面这种写法，肯定就有人这么想：

哇嘎嘎，那就把所有的 tag 全部提前声明，然后就可以按照任意的顺序声明结构体了。

于是，就有了下面这样的代码，

```
typedef struct Polyline_tag Polyline;  ┐
typedef struct Shape_tag Shape;        ├将所有 tag 提前声明
        ⋮                              ┘

struct Shape_tag {
    ShapeType   type;
    union {
        Polyline        polyline;    ←只声明了 tag,使用了 Polyline 的实体!
```

```
        Rectangle       rectangle;
        Circle          circle;
    } u;
};

struct Polyline_tag {
         ⋮
};
```

上面的代码是无法通过编译的。

在只是声明 tag 的情况下，其类型是“不完全类型”。对于不完全类型，**只能取得其指针**（参照 3.2.10 节）。

因为还不知道不完全类型的长度，所以编译器无法确定结构体成员的偏移量。

6.2.4　结构体的嵌套

将结构体作为另一个结构体的成员时，可以像下面这样使用已经声明的结构体：

```
typedef struct {
    int a;
    int b;


} Hoge;

typedef struct {
         ⋮
    Hoge    hoge;    ←—将结构体 Hoge 放到 Piyo 的成员中
} Piyo;
```

也可以声明另一个结构体类型，并且同时将其声明为成员：

```
typedef struct {
    struct Hoge_tag {
        int a;
        int b;
    } hoge;
} Piyo;
```

此处声明的 struct Hoge_tag 在之后还是可以使用的。但是，因为可以省略 tag，所以也可以写成：

```
typedef struct {
    struct {
        int a;
        int b;
    } hoge;
} Piyo;
```

但这种情况下，之后就不能重复使用相同的类型了。

我一般不在结构体声明中再声明结构体，倒是经常在结构体中声明共用体。这个在后一小节中说明。

6.2.5 共用体

共用体几乎总是和结构体、枚举组合使用。

在第 5 章我们定义了下面这样的 Shape 类型，

```
typedef enum {
    POLYLINE_SHAPE,
    RECTANGLE_SHAPE,
    CIRCLE_SHAPE
} ShapeType;

typedef struct Shape_tag {
    ShapeType   type;
    union {
        Polyline        polyline;
        Rectangle       rectangle;
        Circle          circle;
    } u;
} Shape;
```

Shape 可能是 Polyline（多点折线），也可能是 Rectangle（长方形）或者是 Circle（圆）。此时，我们可以用共用体。

为了表示共用体“此时真正使用的成员是哪一个”，使用了枚举型 ShapeType 的变量 type。程序员有责任确保枚举的标识和真正被存储的成员之间的整合性。

在某些书籍当中也会出现下面的共用体用法：

```
typedef union {
    char c[4];
    int int_value;
} Int32;
```

在 int 为 4 个字节的情况下，可以以 C 中规定的字节为单位来访问

int_value 中保存的整数值。

可是，其结果完全依赖于环境的字节排序（参照 2.8 节），并且规范本来也没有规定 int 就是 32 位。

我并没有偏执地认为这种写法是完全错误的，但是在使用这种技巧的同时，的确应该意识到程序的可移植性问题。

6.2.6　数组的初始化

一维数组可以通过下面的方式进行初始化：

```
int hoge[] = {1, 2, 3, 4, 5};
```

因为编译器会去计算数组元素的个数，所以此时没有必要特别地去定义元素个数。为了防止出现不必要的错误，也最好不要在此处定义元素个数。

二维以上的数组，可以像下面这样初始化：

```
int hoge[][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
};
```

除了“最外层”的数组[①]，其他层包含的数组是不能省略元素个数定义的。请参照 3.5.2 节。

6.2.7　char 数组的初始化

char 数组可以通过下面的方式进行特殊的初始化：

```
char str[] = "abc";
```

这其实是

```
char str[] = {'a', 'b', 'c', '\0'};
```

的语法糖。

因为末尾加上了'\0'，所以 str 的元素个数为 4。

此时，多余地加上元素个数的定义，很可能会发生下面的错误：

① 多维数组的“最外层”是指将其他数组作为元素进行包含的最外侧的数组。——译者注

```
char str[3] = "abc";    ←—忘记了'\0'的存在
```

为了避免这样的错误，应该省略元素个数的定义，把对元素计数的工作交给编译器来做。

话说回来，其实在实际编程中，使用字符串初始化的 char 的数组的情况并不多，一般写成下面这样就足够了：

```
char *str = "abc";
```

两者的不同在于：相对于前者初始化“char 的数组”的内容，后者是利用字符串常量初始化“指向 char 的指针”。字符串常量一般保存在只读的内存区域（参照第 2 章），所以后者不能修改字符串的内容*。

* 在有些环境中，也许是可以修改的。这一点毕竟还是要依赖于环境。

6.2.8 指向 char 的指针的数组的初始化

在需要几个字符串组成的数组时，一般我们使用“指向 char 的指针的数组”。

```
char *color_name[] = {
    "red",
    "green",
    "blue",
};
```

最后的 blue 后面加了一个逗号，这并不是一个错误。

C 语言中，在数组的初始化表达式最后的元素后面，既可以加逗号，也可以不加。

很多人讨厌这个规则，我倒是很喜欢。其实在所有的元素后面都添加逗号，还是挺方便的。对于字符串的情况，倘若在最后一个元素后面不追加逗号，当增加一个元素的时候，容易糊里糊涂地写成下面这样，

```
char *color_name[] = {
    "red",
    "green",
    "blue"    ←—忘了加逗号
    "yellow"
}
```

此时，ANSI C 会擅自将相邻的字符串常量连接起来，上面的数组就变成了“red”、“green”、“blueyellow”构成的只有 3 个元素的数组*。

* 很明显，这个问题的起因是“擅自地连接了相邻的字符串”。作为连接字符串的功能，这好像还挺好使的。不过倒不如像 Perl 那样，在字符串之间放入“.”来连接字符串，就不会发生这样的问题了。

Rationale 中“数组的初始化表达式最后的元素后面，可以加逗号，也可以不加逗号”这样的规则，除了使追加/删除元素更为方便之外，还有一个原

因就是，能够更简单地开发自动生成代码的程序（参见 Ratinale 的 3.5.7①）。

但是，枚举的声明中却没有这样的规则，所以我认为这是不完整的规则。

```
typedef enum {
    RED,
    GREEN,
    BLUE,    ←这里不可以加逗号
} Color;
```

现在，一般称为 ANSI C 的 C（ISO-IEC 9899-1990），在语法上是不允许写成上面这样的*。可是在 ISO C99 中，规范已经修改成可以在最后的元素后面加上逗号。

* 根据编译器的不同，有的编译器可能会忽略这种问题，有的编译器会报出警告。

6.2.9 结构体的初始化

假设有下面这样一个结构体：

```
typedef struct {
    int a;
    int b;
} Hoge;
```

写成下面这样，可以初始化结构体的内容：

```
Hoge hoge = {5, 10};
```

在结构体嵌套或者结构体中包含数组的时候

```
typedef struct {
    int         a[10];
    Hoge        hoge;
} Piyo;
```

只要能够像下面这样，很好地将各成员的内容一一对应，也可以完成初始化：

```
Piyo piyo = {
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
    {1, 2},
};
```

6.2.10 共用体的初始化

共用体的初始化比较麻烦。因为编译器无法知道“想要初始化哪一个成员”。

① http://www.lysator.liu.se/c/rat/c5.html#3-5-7

在 ANSI C 中，共用体的初始化是针对**第一个成员**实施的，这还真是个奇怪的规则呢！但是对于 C 的语法来说，这也是迫不得已的决定。

```
typedef union {
    int         int_value;
    char        *str;
} Hoge;
        ⋮
Hoge hoge = {5};    ←—初始化表达式对应于 int_value
```

6.2.11 全局变量的声明

在使用 C 的全局变量的时候，简单地在头文件中像下面这样进行声明：

```
int global_variable;
```

然后可以在使用它的（多个的）`.c` 文件中将其`#include`。这种用法是比较常见的。

可是，本来应该在整个程序的某一个地方**定义**全局变量，其他地方使用 `extern` 声明就可以了。

虽说如此，但是现在的大部分（UNIX 的）C 处理环境中，对于在多个地方进行全局变量的定义，编译器是不会提出任何警告的，这不能不说是 C 的一个缺陷。在多个地方进行不加 `extern` 的变量定义，本来在连接的时候就应该报出**多重定义**（multiple define）的错误。

大型的应用程序，会有很多人同时参与开发。这种情况下，对于偶尔出现的全局变量名称冲突的情况，如今的 UNIX 的处理环境是不会提出任何警告的。最终的结果就是，大家总是被“全局变量的值总是在不知不觉中被修改”这样的性质极其恶劣的 bug 所困扰。

当然，在大型应用程序开发中，建立全局变量命名规则是一个常识。但是运用规则的也是人，所以出现疏漏也是不可避免的。所以应该通过一些工具去机械地检查全局变量的命名状况。

这里说一个题外话。我经常看到某些开发工程中对函数名使用了命名规则，却将全局变量命名规则的制定和实施放在一边。对函数名使用命名规则，这自然是件好事。但如果函数名出现冲突，在连接的阶段，毕竟处理环境是会报错的。因此从危险性上来说，全局变量名称的问题应该更多地引起大家的重视。

顺便说一下，C++已经消除了这个缺陷。如果在多个地方定义没有 extern 的变量会被报错。

在第 5 章的 word_count 程序中，头文件 word_manage_p.h 里使用了 extern 声明了全局变量，然后在 initialize.c 中对它们进行了定义。

可是，在两个不同地方进行几乎完全相同的记述，这很容易成为错误的根源。可以使用下面的方法来解决这个问题，

```
#ifdef GLOBAL_VARIABLE_DEFINE
#define GLOBAL  /* 定义"无" */
#else
#define GLOBAL extern
#endif /* GLOBAL_VARIABLE_DEFINE */

GLOBAL int global_variable;
```

将头文件写成上面这样，然后在程序的某一个地方使用#define 定义 GLOBAL_VARIABLE_DEFINE，并且包含这个头文件，就能保证定义只存在于一个地方，其他地方都是 extern。

使用了这个技巧，就无法使用初始化表达式，所以很多人并不喜欢运用这个技巧。对于全局变量，我自己是很少使用初始化表达式的。因为初始化表达式“只能发生一次作用”，所以我习惯写一个像第 5 章例题中 word_initialize()这样的函数。

但是我们经常需要对数组进行初始化，这种情况下，可以使用下面的方式：

```
GLOBAL char *color_name[]
#ifdef GLOBAL_VARIABLE_DEFINE
= {
    "red",
    "green",
    "blue",
}
    #endif /* GLOBAL_VARIABLE_DEFINE */
    ;
```

征服C指针

本书被称为日本最有营养的C参考书。作者是日本著名的“毒舌程序员”，其言辞犀利，观点鲜明，往往能让读者迅速领悟要领。

书中结合了作者多年的编程经验和感悟，从C语言指针的概念讲起，通过实验一步一步地为我们解释了指针和数组、内存、数据结构的关系，展现了指针的常见用法，揭示了各种使用技巧。另外，还通过独特的方式教会我们怎样解读C语言那些让人“纠结”的声明语法，如何绕过C指针的陷阱。

本书适合C语言中级学习者阅读，也可作为计算机专业学生学习C语言的参考。



图灵社区：www.ituring.com.cn
图灵微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：（010）51095186 转 604

分类建议 计算机/编程语言/C语言

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-30121-5
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks